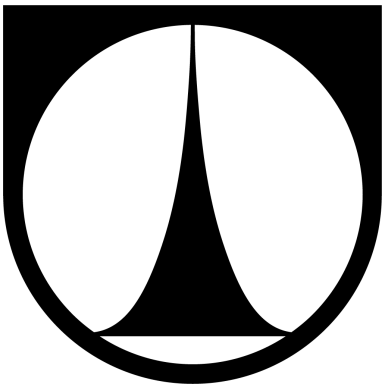
TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií



DIPLOMOVÁ PRÁCE

květen 2013 **Bc. Jiří Sojka**

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: N2612 Elektrotechnika a informatika

Studijní obor: 1802T007 Informační technologie

Klient server aplikace pro Activiti workflow

Client server application for Activiti worflow

Bc. Jiří Sojka

Vedoucí práce: Mgr. Jiří Vraný, Ph.D.

Pracoviště: Ústav nových technologií a aplikované informatiky

Zadání oboustrané, proto prohlášení číslo stránky 5.

Čestné prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum: 16. května 2013

. . . . . .

podpis

Poděkování

Na tomto místě bych chtěl poděkovat především své rodině a nejbližším přátelům, bez jejichž trpělivosti a podpory bych se k této práci zřejmě nedopracoval.

Dále bych zde chtěl poděkovat svému vedoucímu Mgr. Jiřímu Vranému, Ph.D. za ochotu, trpělivost a přístup k vedení mé diplomové práce. Poděkovaní také patří mému konzultantovi panu Ing. Martinu Přádnému, technickému řediteli společnosti Actis s.r.o.

Použitý software

Tato práce byla vysázena programem LaTeX pod operačním systémem Windows 7 64-bit. Nástroj pro tvorbu zdrojových kódů byl použit Eclipse Juno ve verzi 4.2.1 s externími pluginy pro komunikaci s Git repositářem, vytvoření UML diagramů a vygenerování dokumentace ke zdrojovým kódům. Pro uložení dat byly použity MySQL a SQLite databázové servery. Testování proběhlo na aplikačním serveru GlassFish 3.1.2, virtuálním serveru Microsoft Windows Server 2003 a mobilním zařízením Samsung Nexus S s Android 4.1.2.

Kontakt

E-mail: jiri.sojka@tul.cz

Anotace

Tato diplomová práce se zabývá problematikou správy vybraných požadavků z Activiti workflow. Activiti framework je nástroj pro vývoj business aplikací za pomoci procesů modelovaných podle pravidel BMPN 2.0. Cílem práce je navrhnout a implementovat klient server aplikaci pro správu úkolů a procesů.

Práce popisuje návrh a implementaci serveru a dále dvou klientských aplikací. Jedna jako desktopová aplikace v jazyce Java, druhá primárně pro mobilní platformu Android. Komunikace se serverem je realizována využitím REST rozhraní a datového formátu JSON. Data jednotlivých klientských aplikací jsou centrálně synchronizována na serveru poskytujícím webovou službu a komunikujícím s Activiti workflow.

Klíčová slova: Activiti workflow, REST, JSON, webová služba, klient, server, úkol, proces

Abstract

This diploma thesis deals with the issue of administration of selected requirements of Activiti workflow. Activiti Framework is a tool used for development of business applications supported by processes which are modelled according to BMPN 2.0 The main aim of this thesis is to design and implement a client server application for administration of tasks and processes.

This thesis describes a design and implementation of a server and two client applications. One of them is Java (programming) language desktop application and the other one is primarily intended for Android mobile platform. Communication with server is realised by using REST interface and JSON data format. Data of individual client applications are centrally synchronised at the server which provides the web service and communication with Activiti workflow.

Keywords: Activiti workflow , REST, JSON, web service, client, server, task, process

Obsah

Seznam obrázků

Seznam zkratek

AES	Advanced Encryption Standard
API	Application Programming Interface
BPM	Business Process Model
BPMN	Business Process and Notation
EAS	Advanced Encryption Standard
EER	Extended Relationship Diagram
ERD	Entity Relationship Diagram
ID	Identifier
JSON	JavaScript Object Notation
REST	Representational State Transfer
SD	Secure Digital
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
UML	Unified Modeling Language
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
VPN	Virtual Private Network
XML	Extensible Markup Language

1 Úvod

Využívání nových a dostupných technologií je v dnešní době téměř nepostradatelným aspektem úspěšné firmy zabývající se oblastí vývoje softwaru. Aktuálně mají vývojáři možnost vyvíjet aplikace komunikující moderním způsobem pomocí webové služby, dále mohou využívat možností mnoha volně dostupných frameworků, které výrazným způsobem ulehčují a zrychlují tvorbu aplikací komunikujících po síti.

Cílem této práce je úspěšnou implementací návrhu aplikace demonstrovat možnost použití dále popsaných technologií v souvislosti s Activiti *frameworkem*. Ten je v dnešní době společností Actis s.r.o. využíván v mnoha aplikacích.

Aktuální využití Activiti *frameworku* je vázáno na externí aplikaci Lotus Notes [12], kde jsou data ukládána na tzv. dokumenty. Ovšem v případě pádu Activiti *frameworku*, kdy je proveden interními mechanismy přesun *workflow* zpět do posledního konzistetního stavu, již změny dat uložených na dokumentech vráceny nejsou. Tím je porušena datová konzistence, jež se musí následně manuálně a komplikovaně získávat zpět.

Motivací této práce bylo především využít Activiti *framework* jak pro zpracování průběhu vykonávání *procesů*, tak i pro uložení všech potřebných dat aplikace. Tím, že jsou všechna data přímo v Activiti databázi, je zaručena požadovaná konzistence dat.

Základním cílem je možnost spravovat požadavky z Activiti *workflow* pomocí mobilního zařízení se systémem Android. Požadavkem se rozumí správa *úkolů* a start dostupných *procesů*, které jsou uloženy ve formě procesní definice v Activiti databázi.

Konečným cílem práce je vytvoření dvou klientských aplikací s možností zpracování dat v *off–line* režimu. Jedna jako desktopová aplikace v jazyce Java a druhá jako mobilní aplikace pro platformu Android. Komunikace je zprostředkována webovou službou, jež doposud pro komunikaci s Activiti *frameworkem* využívána nebyla. Aktuální stav dat mezi Activiti *frameworkem* a jednotlivými klientskými databázemi je zaručen synchronizačním mechanismem na serveru, který zprostředkovává komunikaci mezi Activiti *frameworkem* a jednotlivými klienty.

Aplikace byla navržena pro případné snadné rozšíření o další datové zdroje. Dále je implementováno jednotné univerzální zobrazení dat a jejich následná validace v klientské části aplikace. Součástí klientské aplikace je také kompletní uživatelský manuál.

Po úvodním seznámení s Activiti *frameworkem* a jeho možnostmi komunikace s externími aplikacemi, které jsou popsány v kapitole 2, je vysvětlen postup návrhu kompletní aplikace, důvody použití centrálního prvku a dále postup návrhu synchronizace dat, která je nezbytná pro udržování aktuálního stavu dat v klientských aplikacích (kapitola 3).

Kapitola 4 již popisuje konkrétní technologie, postupy a řešení problémů, které se vyskytly při samotném vývoji výsledné aplikace.

Součástí práce je také testování a oprava případných nedostatků výsledné aplikace popsané v kapitole 5. Aplikaci je možné rozšířit o možnosti popsané částí 6.

V samotném závěru práce (kapitola 7) jsou zhodnoceny vlastnosti návrhu a implementace aplikace, výsledky testování a především přínos úspěšné implementace pro společnost Actis s.r.o., která je zadavatelem práce.

Jedná se o diplomovou práci, proto bylo k vytváření této zprávy přistupováno s předpokladem znalostí cílové problematiky a práce samotná již popisuje řešení konkrétního návrhu aplikace a její implementace.

1.1 Jazykové konvence

Práce obsahuje přeložené anglické termíny, jejichž pouhý překlad do českého jazyka nepostihuje plnohodnotný význam slova. Některé termíny jsou využívány bez překladu, v českém jazyce pro ně zatím neexistuje ustálený význam. Proto jsou zde uvedeny jejich konkrétní významy:

- *workflow* – označení toku informací v podnikovém procesu a jejich automatizované řízení, v této práci je tento význam vždy spojován s Activiti *workflow*. Využití *workflow* má za účel výrazné zjednodušení vzájemné komunikace a transparentnosti mezi klientem, softwarovým analytikem a samotným vývojářem výsledné aplikace,
- *framework* – softwarová struktura pro podporu vývoje aplikací, v práci tento termín označuje Activiti *framework*,
- *úkol* (user task) – označuje objekt definující uživatelský úkol z Activiti databáze,
- *proces* (process) – označuje objekt procesu určený procesní definicí v Activiti databázi,
- *engine* – označení jádra vykonávajícího základní funkci Activiti *frameworku*.

Pokud je dále v práci některý z těchto termínů označen kurzívou, nabývá tak výše uvedeného významu.

2 Activiti framework

Hlavním cílem této práce je správa *úkolů* a *procesů* v Activiti *workflow*. Tato kapitola popisuje základní vlastnosti a funkce Activiti *frameworku*. Blíže jsou popsány části, které nabízejí přístup pro získání *úkolů*, *procesů* a možnosti jejich zpracování.

Activiti *framework* je označení nástroje nabízejícího vývoj *business* aplikací za pomoci vizualizovaných *procesů* ve *workflow* prostředí.

Projekt Activiti založili v roce 2010 vývojáři Tom Baeyens a Joram Barrez. Jejich cílem bylo vybudovat robustní open-source BPMN 2.0 *process engine*. Již po měsíci vývoje byla k dispozici první stabilní verze. Postupem času vznikla rozsáhlá komunita vývojářů, jejíž součástí se staly i společnosti jako SpringSource, FuseSource nebo Mulesoft, které od této doby vyvíjejí software na základech Activiti. Nespornou výhodou při vývoji je možnost využít flexibilní komunikace se samotnými vývojáři tohoto *frameworku*.

Především podle požadavků vývojařů využívajících tento produkt je Activiti *framework* stále vyvíjen a zdokonalován. Aktuální verze nese označení 5.12. Podle změn v samotném jádru *frameworku* jsou také aktualizovány funkce a vlastnosti Activiti Modeleru a Activiti Designeru, proto je pro vývojáře využívající Activiti *framework* výhodou sledovat aktuální vývoj *frameworku* a používat jeho nejnovější nabízené funkce.

Jádro *frameworku* je napsáno v jazyce Java a šířeno jako open–source pod licencí Apache [4]. Activiti je možno využívat buď jako plnohodnotnou aplikaci, nebo jako součást komplexnějšího projektu jak na serveru, tak například na clusteru, nebo v dnešní aktuálně se rozšiřující technologii zvané *cloud*.

2.1 Součásti frameworku

Obecný pojem workflow obvykle popisuje technologii řízení podniku, nebo například zpracování dokumentů, a jedná se o schéma provedení komplexnější činnosti (procesu). Obecně platí, že je workflow tvořeno především pravidly definující procesy, metrikami procesů a předávanými daty. Tato data jsou předávána v podobě tzv. procesních proměnných, které mohou být definovány již v samotné procesní definici. *Procesy* jsou navrhovány a modelovány podle pravidel BPMN 2.0.

2.1.1 BPMN 2.0

Soubor principů a pravidel BPMN, aktuálně ve verzi 2.0 [11], slouží pro grafické znázornění *procesů* v procesním diagramu. Tento diagram je založen na tzv. flowchart technologii a graficky znázorňuje jednotlivé kroky algoritmu. Jeho výsledná prezentace je podobná diagramu activit z jazyka UML.

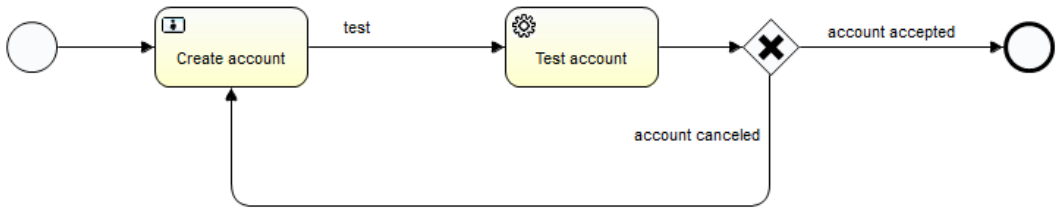
BPMN si klade za cíl stát se jedinou univerzální notací pro tvorbu modelů především právě podnikových procesů. Za tímto účelem je základním stavebním prvkem jazyk XML. Výsledkem by měl být jednotný a konzistentní jazyk.

2.1.2 Jádro frameworku

Obecnou funkcionalitu jádra *frameworku* (*enginu*) je možné si představit jako systém, který zajišťuje změnu stavu v *procesu* dle jeho definice v Activiti databázi. Definice procesu podle pravidel BPMN 2.0 obsahuje:

- events – události, například start a ukončení *procesu*,
- tasks – úkoly označující konkrétní stavy *procesu*,
- gateways – brány, které spojují jednotlivé toky,
- sequence flow – toky popisující přechody mezi jednotlivými stavy procesu.

Následující schéma zobrazuje ukázku *procesu* a jeho částečnou definici v jazyce XML.



Obrázek 1: BPMN proces

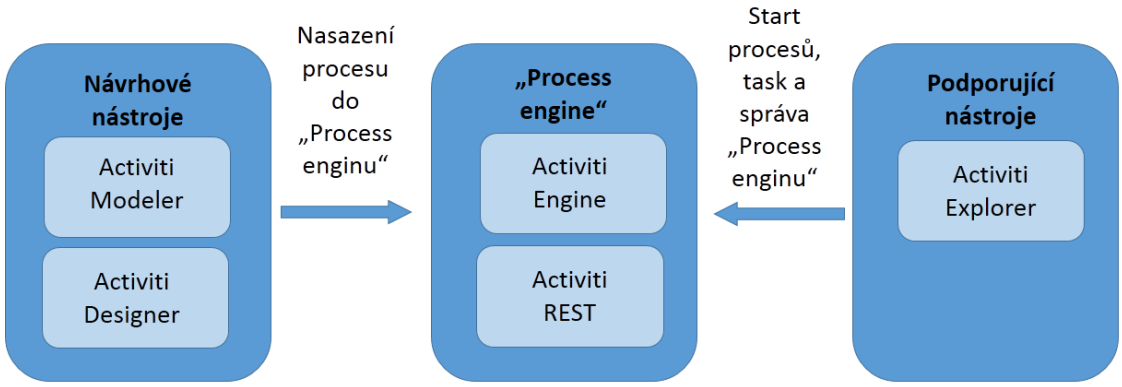
Příklad části procesu v jazyce XML:

```
<process id="create_new_account" name="Create new account">
   <startEvent id="start_process" name="Start"></startEvent>
   <userTask id="create_account" name="Create account">
    <extensionElements>
      <taskListener event="create" class="NewAccount"></taskListener>
    </extensionElements>
   </userTask>
```

Celý *framework* obsahuje sadu komponent, jejíž hlavní součástí je *process engine*. Toto jádro *frameworku* poskytuje základní funkce pro start *procesu*, přechod mezi definovanými stavy či jeho ukončení. Dále obstarává *deploy* (nahrání) nových procesních definic a start procesních instancí.

Activiti komponenty:

- Engine – základní komponenta *frameworku*, zajšťuje spuštění a průběh BPMN procesu,
- Modeler – webové modelovací prostředí pro tvorbu BPMN 2.0 procesů,
- Designer – plugin pro vývojové prostředí Eclipse, který slouží k modelování a definování procesů,
- Explorer – webová aplikace pro správu Activiti *workflow*. Umožňuje startovat nové *procesy*, spravovat uživatele, vizualizovat samotnou Activiti databázi,
- REST – webová služba poskytující REST API pro komunikaci s *enginem* z externích aplikací.



Obrázek 2: Schéma Activiti frameworku

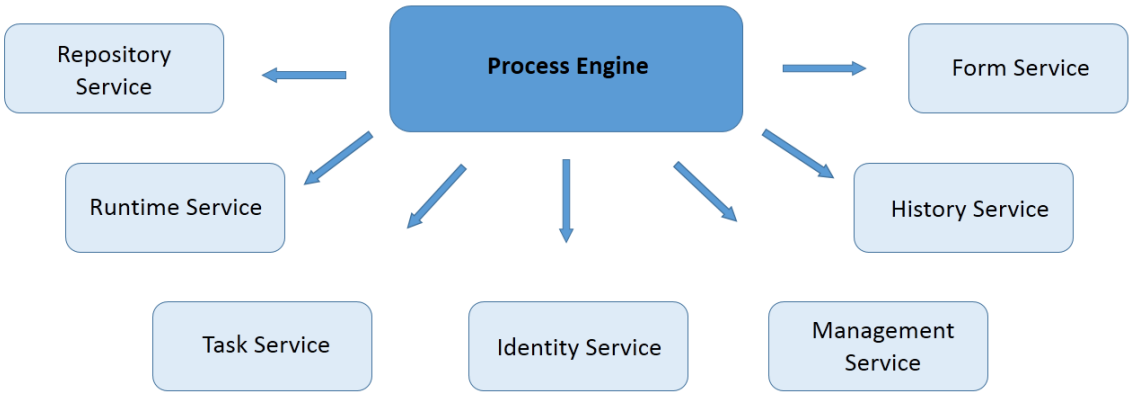
Activiti *framework* nabízí dvě možnosti aplikačního použití:

- *embedded* – aplikace přímo obsahuje Activiti JAR a využívá Java API pro přístup k Activiti *enginu*. Je nutné *engine* inicializovat při každém spuštění aplikace pomocí konfiguračního XML souboru,
- *standalone* – Activiti *engine* je instanciován pouze v aplikaci na serveru. Ostatní aplikace mají možnost přístupu k *enginu* přes tzv. REST rozhraní.

2.2 Activiti engine API

Activiti API [20] pro ovládání *enginu* je rozděleno do sedmi základních částí. Pro vývojáře je výhodné seznámit se nejprve s architekturou Java API. Poté je mnohem snažší pochopit případné ovládání *enginu* přes REST API.

Přístup k jednotlivým částem *enginu* je zřejmý z následujícího diagramu.



Obrázek 3: Základní architektura Process Enginu

Centrální částí je třída *ProcessEngine*. Z této instance lze získat všech 7 základních instancí pro kompletní ovládání Activiti *enginu*. Získání základní instance *process enginu* je implementováno:

```
ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
```

Po získání této instance má vývojář snadný přístup ke všem potřebným částem *enginu*:

```
TaskService taskService = processEngine.getTaskService();
```

2.3 Activiti REST API

Activiti nabízí komunikaci s Activiti *enginem* distribuovaným způsobem přes REST rozhraní. Díky podpoře technologie REST, ve všech aktuálně nejpoužívanějších programovacích jazycích, je volba implementace vzájemné komunikace mezi aplikací a Activiti pouze na samotném vývojáři.

Při komunikaci s REST rozhraním je kvůli bezzstavovosti protokolu HTTP potřeba autentizovat uživatele. Ověření je realizováno pomocí HTTP hlavičky, ve které je přenášeno uživatelské jméno a heslo. REST rozhraní při každém požadavku ověřuje uživatele a heslo ve své vlastní databázi. Po úspěšném ověření je odeslána odpověď s požadovanými daty, případně chybové ohlášení o neúspěšné autentizaci.

URI REST služby je rozděleno v závislosti dle ekvivalentího použití klasického Java API. Podle defaultního nastavení REST služby je základní část URI pro komunikaci s *enginem*:

```
http://IP_adresa:port/activiti-rest/service/
```

Za toto URI jsou intuitivně přidávány části URI pro získání požadovaných dat. Pro získání kompletního seznamu *procesů* z Activiti databáze, je vytvořen HTTP GET požadavek s autentizační hlavičkou pro:

```
http://IP_adresa:port/activiti-rest/service/process-definitions
```

Odpověď na tento požadavek je seznam *procesů* uložených v Activiti databázi. Odpověď je ve formátu JSON:


```
{
  "data": [
    {
      "id": "create_new_account:1",
      "key": "new_account",
      "version": 1
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```


Přidáním `?size=2&order=desc` k původní části URI je možno získat setříděnou část požadovaných dat.

Ekvivaletní získání dat z *enginu* za pomoci Java API je následující:
`repositoryService.createProcessDefinitionQuery().desc().listPage();`

2.4 Základní použití frameworku

Samotné použití Activiti *frameworku* je rozděleno na několik částí. První z nich je návrh a vytvoření *procesu* například v Actviti Designeru podle pravidel BPMN 2.0. Hotový předpis *procesu* (procesní definice) je poté potřeba nahrát do Activiti databáze - provést tzv. *deploy* procesní definice buď využitím Java API, nebo pomocí webové aplikace pro správu *frameworku* - Activiti Explorer.

Informace o *procesu* jsou uloženy v tabulce v Activiti databázi pod jedinečným identifikátorem. Ten obsahuje také číslo poslední verze *procesu*. Z této tabulky je tedy možný výběr procesní definice pro start vykonávání *procesu* tzv. *execution*. Po startu *execution* je *proces* spuštěn a vytvořen první *úkol* dle konkrétní definice (také je možnost startu více *úkolů* nebo v extrémním případě žádného).

Execution jsou uložena ve vlastní tabulce, kde jsou uchována potřebná data pro vykonávání *procesu*. Tím se rozumí především ID procesní definice, podle které je *proces* vykonáván, dále aktuální ID stavu, ve kterém se *proces* nachází a několik dalších vlastností definující stav vykonávání, jako například zda je *execution* v aktivním stavu.

Průběh *procesu* je realizován přechodem mezi definovanými stavy (*úkoly*). V Activiti Designeru je možno definovat události právě například na start nového *úkolu* nebo

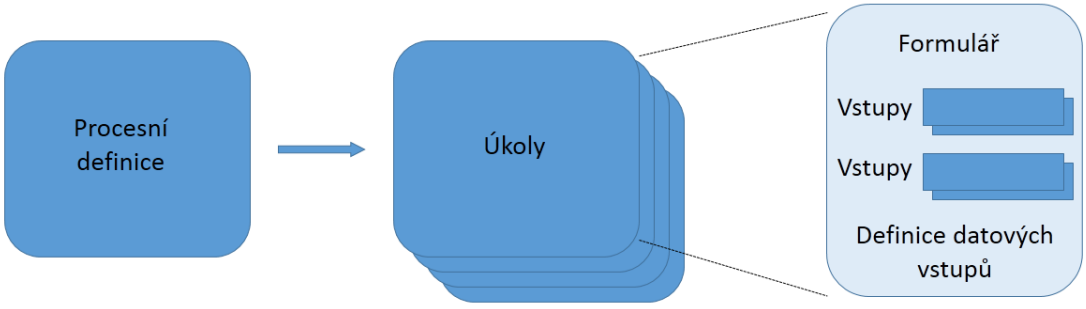
na jeho ukončení. Každý objekt *úkolu* má v tabulce své jedinečné ID, ID *procesu*, *execution* ID pod kterou je *úkol* vykonáván, a dále informace o právech (vlastník nebo uživatel, který má právo na ukončení konkrétního *úkolu*).

Průběh přechodů v procesu mezi stavy je realizován na základně hodnot procesních proměnných. *Workflow* podle hodnoty procesních proměnných rozhodne, kterou cestou bude vykonávání *procesu* dále pokračovat (například rozhodnutí na *gateway*). API poskytuje několik operací s *úkoly*, jedná se především o následující:

- *claim* – převzetí řešení *úkolu*,
- *unclaim* – odhlášení se od řešení *úkolu*,
- *complete* – dokončení *úkolu*.

Při procesu dokončení *úkolu* je možné (podle procesní definice případně vyžadováno) předat konkrétní hodnoty procesních proměnných, podle kterých se *workflow* řídí.

Pro komunikaci s uživatelem využívá Activiti tzv. formuláře (*forms*) a jejich vlastnosti (*properties*). Vlastnosti formuláře jsou generovány podle procesních proměnných z definice konkrétního *úkolu* a jejich získání je možné použitím standartního API. Výhody použití formulářů jsou především jednotná komunikace s uživatelem či možnost kontroly formátu a typu vstupních dat.



Obrázek 4: Generování formuláře

Mezi vlastnostmi formulářů jsou kromě ID a názvu také požadovaný datový typ a příznak, zda je vyplnění formulářového pole pro uživatele povinné, či ne.

Vykonávání *procesu* je tedy vždy start *úkolu* a přiřazení řešitelů. Uživatel, který má práva na dokončení *úkolu*, ho ukončí s případnými požadovanými parametry a *workflow* rozhodne, který *úkol* bude nastartován jako následující.

Takto probíhá *proces* od tzv. startovní události (*start event*) až po událost konečnou (*end event*).

2.5 Správa task z pohledu uživatele

Activiti REST rozhraní nabízí pro správu *úkolů* základní možnosti: *complete, claim a unclaim*. Každý *úkol* může mít definována práva pro případné zpracování uživatelem. Uživatel má možnost *úkol* zpracovat, pokud je jeho uživatelský identifikátor v:

- *assignee* – pole označující uživatele, kterému je *úkol* přidělen,
- *candidate* – pole označující případné kandidáty na zpracování,
- *candidate–group* – pole určující skupiny uživatelů, kteří mají právo na zpracování.

2.6 Webová služba

Jedna z možností komunikace s Activiti je realizována pomocí webové služby. Webová služba označuje systém pro součinnou spolupráci strojů na síti. Služba samotná je prezentována softwarovým nebo hardwarovým agentem. Schéma komunikace je vždy stejné, jeden agent o službu žádá a druhý ji poskytuje. Standardy používané webovými službami zajišťují totožnou sémantiku obou agentů.

Přínosem webových služeb je možnost integrace libovolných aplikací provozovaných na různých platformách a možnost jejich ovládání prostřednictvím webového rozhraní za použití vzdáleného volání procedur přes síť. V dnešní době jsou využívány dvě hlavní architektury pro komunikaci webových služeb: SOAP a REST.

2.6.1 SOAP

SOAP [24] je označení protokolu pro výměnu zpráv definovaných v jazyce XML, především pomocí protokolu HTTP. Právě díky protokolu HTTP má SOAP možnost průchodu přes firewall, kde bývá protokol HTTP zpravidla povolen. SOAP tvoří základní vrstvu pro komunikaci mezi webovými službami a poskytuje prostředí pro implementaci složitější komunikace.

SOAP definuje strukturu a formát zprávy pomocí XML. Tento formát je využíván především z důvodu dostupnosti různých parsovacích nástrojů v mnoha programovacích jazycích. XML formát disponuje delší syntaxí, což zvyšuje čitelnost zprávy pro samotného uživatele a umožňuje následnou snadnou validaci obsahu. To umožnuje předejít chybám při vzájemné komunikaci, ale také zvyšuje paměťovou náročnost, více procesorového času a především zvyšuje režii přenosu. Tím pádem je snížena rychlost komunikace.

2.6.2 REST

REST je označení architektonického stylu rozhraní navrženého pro distribuované prostředí. Rozhraní REST je používané pro jednotný přístup ke zdrojům (takzvaným resources), přičemž data mohou mít odlišnou reprezentaci (ATOM, XML, JSON). Dnešní moderní frameworky pro vývoj server-side aplikací podporují tvorbu REST rozhraní a dokaží definovat procedury pro všechny potřebné metody, což vývojářům výrazně ulehčuje používání REST rozhraní.

Aby mohl být systém označován pojmem REST, je nutné, aby splňoval šest základních vlastností vycházejících ze základní myšlenky jeho použití [3] - *klient - server architektura*, *bezstavovost*, *využití cache paměti*, *kód na vyžádání*, *vrstvený systém* a *jednotné rozhraní*.

REST architektura je na rozdíl od SOAP orientována datově, nikoliv procedurálně. Zdroje (*sources*) musí vlastnit jedinečný identifikátor URI a REST definuje čtyři základní metody pro přístup k datům.

2.6.3 CRUD

Základní metody pro přístup k datům jsou vytvoření dat (CREATE), získání konkrétních dat (RETRIEVE), změna dat (UPDATE) a konečně smazání dat (DELETE). Tento přístup je implementován dle základních metod aplikačního protokolu HTTP.

Implementace CRUD skrze HTTP:

- *CREATE* – pro vytvoření dat slouží metoda POST,
- *RETRIEVE* – metoda pro získání zdroje, implementována pomocí metody GET,
- *UPDATE* – změna již existujících dat implementována metodou PUT,
- *DELETE* – smazání zdroje HTTP metodou DELETE.

Activiti *framework* a jeho vrstva zajišťující REST rozhraní jsou stále vyvíjeny a tím jsou rozšiřovány možnosti využití webové komunikace s tímto *frameworkem*.

2.7 Zhodnocení aktuálního stavu

Zadání práce spočívá ve vytvoření klient–server aplikace pro Activiti *workflow*. Po upřesnění požadavků je cílem práce vytvořit vlastní server sloužící pro synchronizaci dat a dvě klientské části. Tyto klientské části se dělí na desktopovou, jejíž účel je především testování komunikace, a na naitivní aplikaci pro platformu Android.

Klientská aplikace pro platformu Android komunikující s Activiti *workflow* je dostupná na Google Play [8] pod názvem *Activiti Explorer Beta*. Označení *beta* je vhodné,

i když popis aplikace slibuje základní funkcionalitu, po zadání přihlašovacích údajů se aplikace, zřejmě kvůli interní chybě, ukončí. Z tohoto důvodu nebylo možné otestovat možnosti stávající aplikace. Dle dostupného popisu také aplikace nepodporuje možnost práce v tzv. off–line režimu, což bylo jedním z požadavků na výslednou aplikaci této práce.

Celkové zadání se řídí požadavky zadavatele, tedy společností Actis s.r.o. Návrh komunikace, ve které je centrálním prvkem server obsluhující synchronizaci dat a zároveň klient pro Activiti *workflow*, doposud tato společnost nevyužívá. Úspěšná implementace je zároveň důkazem možnosti použití dále popsaných technologií.

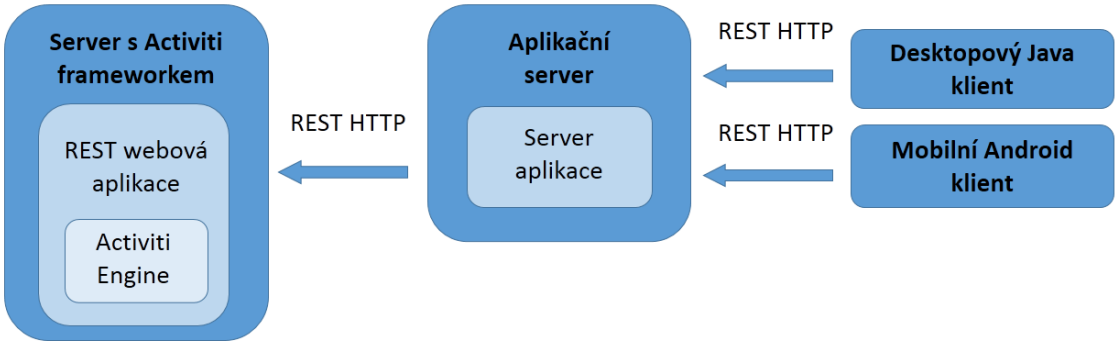
Obě klientské části popsané v následujících kapitolách, jsou navrženy a implementovány dle rozhraní vlastního serveru. Tato vlastnost zaručuje, že jsou klientské aplikace jedinečným řešením cílové problematiky.

3 Návrh aplikace

Cílem práce je navrhnout a implementovat aplikaci pro správu vybraných požadavků z Activiti *workflow*. Klientská část aplikace by měla být implementována především jako nativní aplikace pro platformu Android s možností práce s daty i bez internetového připojení (tzv. off–line režim). Vybranými požadavky jsou nastartování *procesu* dle procesní definice uložené v Activiti databázi a správa *úkolů*, na které má uživatel právo (task manager). V této kapitole je popsán způsob komunikace, návrh databáze a synchronizace dat mezi klientem a serverem.

Prvotním návrhem od zadavatele, který měl být i zadáním práce, byl návrh a implementace pouze klientské části pro platformu Android. Komunikace s Activiti *frameworkem* včetně autentizace klientů měla být implementována pomocí REST rozhraní. Data by byla na straně klienta uchována v SQLite databázi. Po konzultaci tohoto návrhu byla zvážena možnost využití tzv. centrálního prvku, který by umožňoval klientům komunikaci s více zdroji informací.

Tento návrh byl postupně upravován, především kvůli splnění nových požadavků na možnosti aplikace. Těmi byly snadná možnost změny autentizace uživatelů, případná snadná a rychlá reakce na změnu Activiti REST rozhraní. Pro splnění všech požadavků byl vytvořen finální návrh, ve kterém je centrálním prvkem reprezentován vlastní server s webovou službou.



Obrázek 5: Návrh implementace

Server bude umožňovat klientům správu požadavků ve více zdrojích (databázích) prostřednictvím jednoho připojení. Dále bude umožněna snadná změna autentizace na jednom místě. Klientská část aplikace vždy odesílá autentizační data v pevně definovaném formátu, způsob autentizace již bude zprostředkovávat server. Záleží na samotné konfiguraci serveru, zda ověří uživatele oproti Activiti *frameworku*, nebo případně oproti LDAPu. Server slouží jako prostředník pro komunikaci mezi klienty a Activiti *frameworkem*, proto se reakce na případnou změnu Activiti REST API týká pouze serverové části.

Za účelem možnosti využití klientské aplikace také na počítači (nejen emulátor platformy Android) a pro snadnější možnost testování vzájemné komunikace mezi klientem a serverem je v požadavcích práce implementace dvou klientských částí. Klientská aplikace implementována pro platformu Android a druhá jako desktopová aplikace v jazyce Java. Požadavky na aplikaci jsou snadná rozšířitelnost pro případné obecnější využití a využití možností Activiti *frameworku* bez zásahu do jeho aktuální implementace.

3.1 Návrh komunikace

Po kompletním návrhu aplikace na nejvyšší úrovni abstrakce (obrázek 5) je potřeba zvážit a určit možnosti samotné implementace. Jedním ze stěžejních faktorů je volba implementace komunikace.

Komunikací mezi serverem a klientem byla zvolena webová služba využívající REST rozhraní. Webová služba byla vybrána především pro snadnou rozšiřitelnost v případě potřeby, podporu vývojářských frameworků a využitelnost ve většině dnešních moderních aplikací.

Datový komunikační model ve formátu JSON [2] poskytující Activiti *framework* musí být zpracován na straně serveru. Server musí být schopen data nejen přijmout, ale také odesílat (autentizační informace o uživatelích, hodnoty procesních proměnných pro dokončení *úkolů* atd.). Proto je vhodným řešením využít alespoň část tohoto datového modelu ve formátu JSON také pro komunikaci mezi serverem a klientem. Formát JSON má (podle dostupného porovnání od vývojářů z *MyDevNotes* [15]) kratší délku než formát XML, což je z hlediska objemu přenášených dat výhodné. Pokud je tento formát využíván Activiti webovou službou, bylo by zbytečné na straně serveru převádět formát JSON například do formátu XML. Další a neméně významnou výhodou je podpora formátu JSON jak v jazyce Java pro vývoj desktopových aplikací, tak i v jazyce Java pro vývoj aplikací na platformě Android.

3.2 Návrh databáze

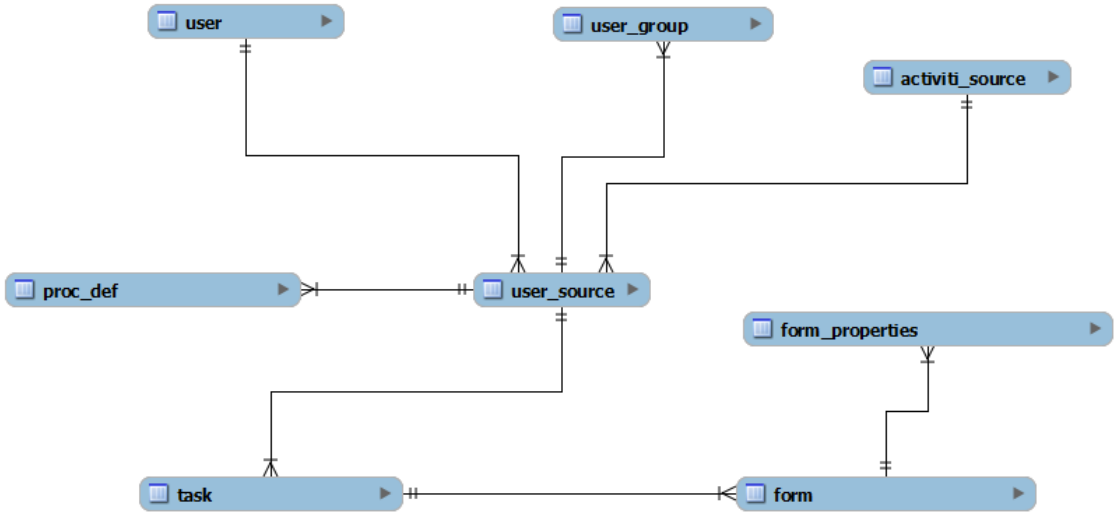
Z požadavku na možnost centrálně spravovat data z více Activiti databází a následnou synchronizaci s klientem vychází určité nároky na uložení dat. Data je nutné uchovávat jak na straně serveru, tak na straně klienta pro možnost práce s aplikací v off–line režimu. Jako databázový systém na straně serveru a desktopové klientské části v jazyce Java byl pro svoji licenci, multiplatformnost a také využití v jiných projektech zvolen server MySQL 5.6. [18].

3.2.1 Server databáze

Na straně serveru je potřeba uchovávat data o URL webových službách jednotlivých Activiti *frameworků*. Dále data o uživatelích, kteří mají přístup a možnost spravovat *úkoly* a *procesy* v těchto Activiti databázích.

Správa *úkolů* a start nových *procesů* definují potřebu ukládat a synchronizovat datové objekty obsahující veškeré informace o *úkolech* a procesních definicích. Dále pro možnost dokončení *úkolu* a vyplnění hodnot případných procesních proměnných jsou uchovávány formuláře a jejich vlastnosti.

Z těchto požadavků byl vytvořen následující ERD model databáze na straně serveru zachycující jednotlivé vazby mezi entitami.



Obrázek 6: ERD model server databáze

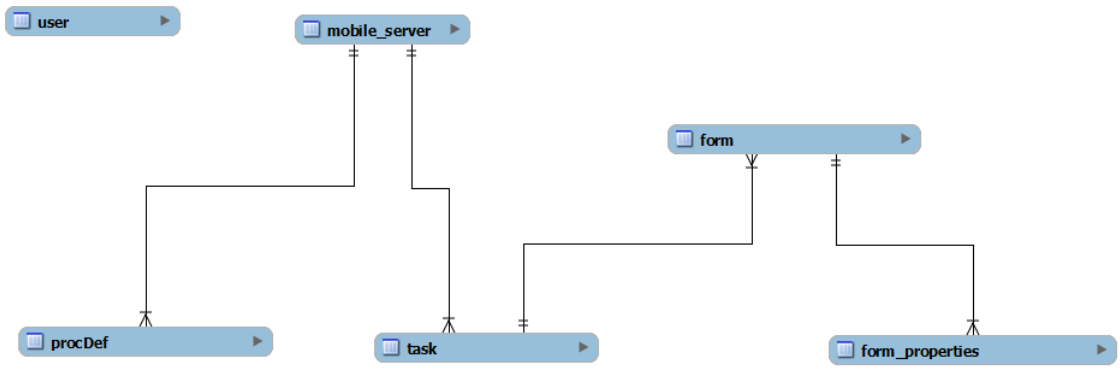
Atributy entit *úkol* (task), *proces* (proc_def), formulář (form) a formulářová vlastnost (form_properties), jsou určeny již z Activiti databáze. Pro jejich jednoznačnou identifikaci v databázi jsou tyto entity rozšířeny o přirozený klíč. Dále jsou uloženy informace o klientech a skupinách, ve kterých je konkrétní uživatel členem.

Databáze by v případném rozšíření aplikace mohla sloužit také jako centrální zdroj informací například pro statistické přehledy využívání jednotlivých Activiti webových služeb. Výhodou komunikace klientů přes server je možnost centrální modifikace a validace uložených dat, případně získání dat z jiného datového zdroje.

3.2.2 Klientská databáze

Požadavkem na klientskou aplikaci bylo také její využití v takzvaném off–line režimu (bez připojení k internetu). Tímto požadavkem je nutné uchovávat data nejen na serveru, ale také v samotné klientské části aplikace. Uložení dat musí být navrženo

tak, aby klient mohl zpracovávát požadavky a tyto změny zaznamenávat pro následnou synchronizaci v on–line režimu.



Obrázek 7: ERD model klientské databáze

Pro uložení dat v klientské části pro platformu Android byl využit interní databázový systém SQLite [21]. MySQL Workbench umožnuje snadný export EER modelu do SQL skriptu. Pro převod tohoto návrhu do SQLite skriptu byl využit online convertor *MySQL to SQLite* [14]. Výsledek konvertoru bylo potřeba dodatečně manuálně upravit dle SQLite dokumentace. Desktopová aplikace v jazyce Java využívá stejně jako server databázi MySQL ve verzi 5.6.

Výsledný ERD model klientské databáze je, co se týče samotných dat (*proces*, *úkol*, formulář), téměř totožný s modelem databáze na serveru. Rozdílné jsou primární klíče jednotlivých tabulek. Klientská aplikace je od začátku navrhována jako jednouživatelská, není tedy nutné uchovávat data o více uživatelích ani jejich identifikátor používat jako součást klíče. Dále je v klientské databázi přidána tabulka *user*, ve které jsou uchovávány informace o konkrétním uživateli. Tato tabulka slouží pro off–line přihlašování a uchování stavu aplikace.

3.3 Synchronizace dat

Pokud jsou klientská data uchovávána na straně serveru a zároveň má každý uživatel svá data ve své klientské databázi (pro možnost práce v off–line režimu), je nutné tato data synchronizovat a udržovat tak v aktuálním stavu jak na serveru, tak v klientské části aplikace.

Synchronizace dat znamená získání nových *úkolů* a *procesů* pro konkrétního klienta. Samozřejmě také smazání již vyřešených *úkolů* a *procesů*, které se již v Activiti databázi nevyskytují. Během synchronizace je proveden případný start *procesů* či dokončení *úkolů*. Při návrhu synchronizace je nutné být seznámen s postupem vykonávání určitých kroků v Activiti *frameworku* a určit prioritu jednotlivých částí synchronizace.

3.3.1 Prvotní návrh synchronizace

Definice procesu je uložena v databázi a *úkoly* postupně vznikají při vykonávání samotného *procesu*. Detailnější popis základní funkce Activiti *frameworku* je uveden v kapitole *Základní použití frameworku* (2.4). Základní funkcionalita *frameworku* definuje určité vlastnosti:

- *proces* - je určen definicí procesu v Activiti databázi,
- *úkol* - vzniká při startu procesu nebo po dokončení předchozího *úkolu*, jeho definice vychází z definice procesu.

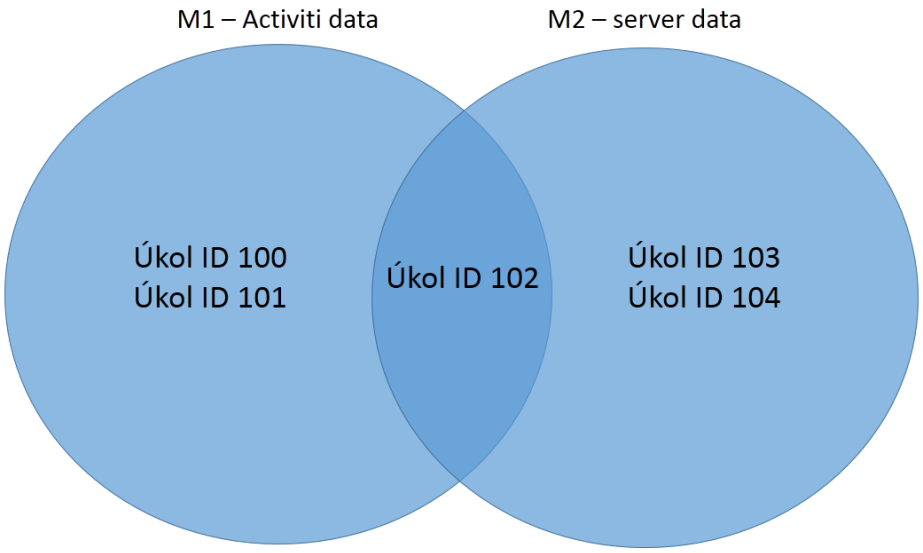
Tyto vlastnosti určují seřazení všech 4 kroků synchronizace mezi serverem a Activiti *frameworkem*. Z klientského hlediska má vyšší prioritu start nových *procesů* a dokončení požadovaných *úkolů*. Až poté je dle priority jejich synchronizace. Toto pořadí je také logicky určeno vznikem nových *úkolů* až po dokončení předchozího *úkolu* nebo startem nového *procesu*. Výsledné kroky synchronizace jsou následující:

- 1. krok – nastartování *procesů*,
- 2. krok – dokončení všech požadovaných *úkolů*,
- 3. krok – synchronizace *procesů*,
- 4. krok – synchronizace *úkolů*.

Samotná synchronizace dat má být po konzultaci se zadavatelem realizována pomocí základních operací s množinami. Při synchronizaci *úkolů* si server vyžádá pro konkrétního klienta všechny *úkoly* ze všech Activiti databází. Tato data tvoří jednu množinu M1 (viz obrázek 8). Dále získá z vlastní databáze všechny stávající *úkoly* pro konkrétního uživatele - druhou množinu M2.

Nyní pokud provedeme množinové odečtení M1 - M2, získáme podmnožinu, jež tvoří *úkoly*, které doposud server ve své databázi nemá, tudíž se jedná o *nová data* (ID 100, 101).

Pokud provedeme opačné odečtení M2 - M1, získáme tak podmnožinu, kterou tvoří *úkoly*, které se nevyskytují v žádné Activiti databázi (ID 103, 104). To znamená, že tato data již byla smazána a můžeme je označit jako *stará data*. Stejný postup je možné použít i při synchronizaci *procesů*.



Obrázek 8: Znázornění množin

Touto částí je vyřešena synchronizace mezi serverem a Activiti databázemi. Dále zbývá navrhnout synchronizaci dat mezi databází serveru a klienta. Návrh na synchronizaci se skládal z následujících kroků:

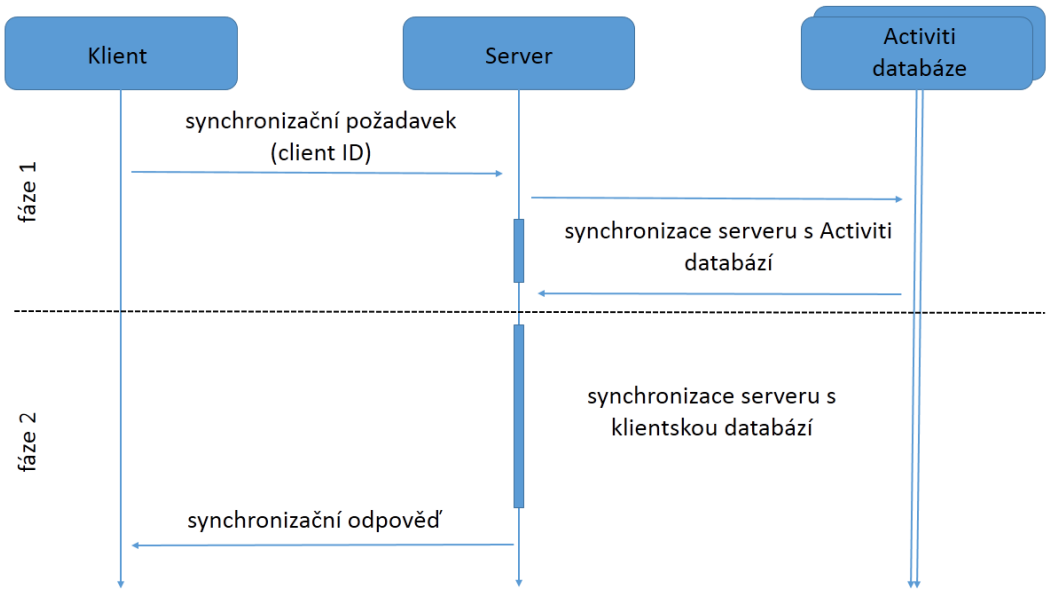
1. dle klientského ID synchronizovat data mezi serverem a Activiti databází,
2. nová data uložit na serveru a zároveň je odeslat klientské aplikaci s označením nových dat,
3. stará data ze serveru smazat a zároveň je odeslat klientské aplikaci s označením starých dat.

Tento postup byl demonstrativně implementován a otestován. Hlavním nedostatkem testování se ovšem prokázala nemožnost sesynchronizovat data s více klientskými aplikacemi, které používá jeden uživatel. Server měl v tuto chvíli ve své databázi stejná data jako jedna konkrétní klientská aplikace. Pří vývoji druhé klientské části a požadavku na používání obou dvou klientských aplikací jedním uživatelem zároveň (mobilní aplikace i desktopová aplikace využívána zároveň jedním uživatelem) bylo nutné navrhnout a implementovat sofistikovanější řešení, jež by umožňovalo nezávislou synchronizaci s více klientskými aplikacemi.

Při synchronizaci *úkolů* jsou vždy současně synchronizovány i ostatní data s nimi spojená – formuláře a jejich vlastnosti. Při ukládání nových *úkolů* do databáze jsou vyžádána a uložena také formulářová data. Současně se smazáním *úkolů* jsou kaskádově smazána i všechna formulářová data, která jsou na *úkol* v databázi vázána cizím klíčem. Tento mechanismus je použit jak na serveru, tak i v klientské části aplikace.

3.3.2 Konečný návrh synchronizace

Výše uvedený návrh neřeší kompletní problematiku uložení dat na straně serveru a synchronizaci, proto byl návrh upraven do následující podoby.



Obrázek 9: Schéma konečného návrhu synchronizace

Server uchovává aktuální data klientů v databázi. Každá z klientských aplikací ovšem může mít data ve své databázi různá. Synchronizaci je tedy nutné provádět s každým klientským zařízením individuálně. Proto byla ve výsledku navržena synchronizace skládající se ze dvou částí. První část slouží k synchronizaci mezi serverem a Activiti databázemi, ve druhé části jsou sesynchronizována data mezi serverem a konkrétním klientem. Klient tedy ve svém požadavku zasílá serveru informace o datech, které již vlastní ve své databázi.

V první fázi (fáze 1) na obrázku 9 odesílá klient požadavek na synchronizaci (podrobný popis požadavku a odpovědi v příloze 1.5). Server podle klientského ID sesynchronizuje vlastní data s Activiti databází (s jednou či více dle konfigurace). Po tomto kroku jsou data na serveru v aktuální podobě.

Ve druhé fázi (fáze 2) dochází k synchronizaci dat mezi klientem a aktuálními daty na serveru. Klient již ve svém úvodním požadavku zasílá informaci o datech, která má ve své vlastní databázi. Podle těchto údajů jsou data výše popsaným způsobem (práce s množinami) sesynchronizována a v posledním kroku server odesílá odpověď se stavem synchronizace, *novými* a *starými* daty.

3.4 Využití moderních programovacích prostředků

V souladu s moderními přístupy k návrhu aplikace v objektovém jazyce byly využity při implementaci některé z návrhových vzorů nebo jejich částí.

Návrhové vzory nabízí svojí koncepcí snadné řešení typických programátorských problémů a zvyšují efektivitu práce. Nejedná se přímo o části zdrojových kódů, ale o postup, jak řešit daný problém. Objektové návrhové vzory využívají vlastností objektově orientovaných jazyků, jako jsou rozhraní (interface) či polymorfismus. Problematika a příklady praktického využití návrhových vzorů jsou obsaženy v knize *Návrhové vzory* od pana Pecinovského [19].

Konkrétní využití návrhových vzorů v této práci je popsáno v kapitole 4.2.6 a 4.2.7.

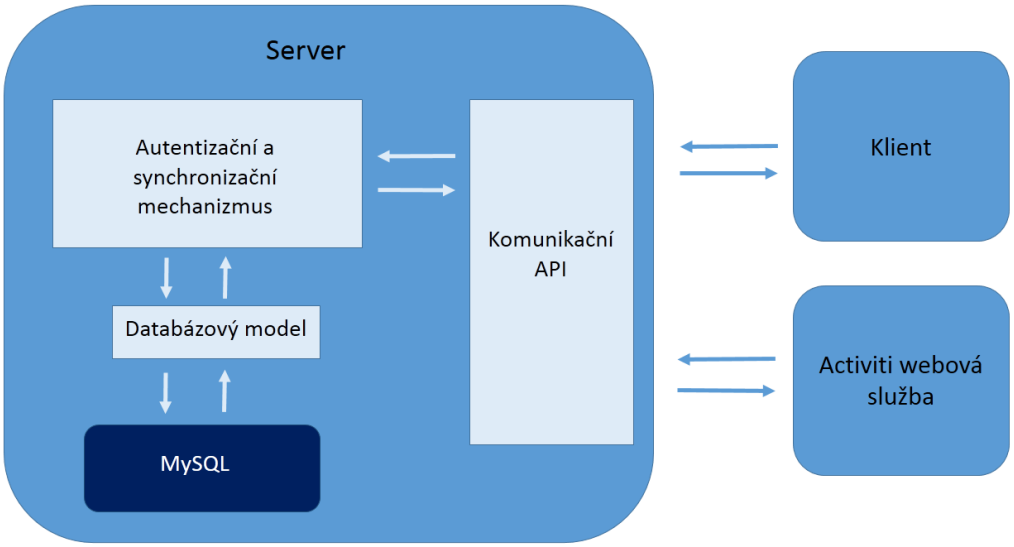
4 Implementace

Po kompletním návrhu aplikace popsané v kapitole 3 se podařilo tento návrh implementovat. Tato kapitola popisuje klíčové postupy, zvolené implementační prostředky a metody samotné implementace jak serverové části aplikace, tak i desktopové a mobilní klientské aplikace.

Cílem projektu bylo vytvořit aplikaci, která by byla snadno rozšiřitelná nejen o REST rozhraní, ale také o případné zpracování a synchronizaci dat z jiného datového úložiště.

4.1 Server

Serverová aplikace s webovou službou využívající REST rozhraní (2.6.2) byla implementována na aplikačním serveru GlassFish 3.1.2 [16] v jazyce Java. Vývojové prostředí Eclipse Juno nabízí po instalaci pluginu přímou správu tohoto aplikačního serveru a urychluje tím vývoj a případné testování aplikace.



Obrázek 10: Základní schéma server části

Samotná implementace byla rozdělena do několika kroků:

- komunikace – implementace REST rozhraní pro komunikaci jak s Activiti *frameworkem*, tak s klientskými aplikacemi,
- synchronizace – implementace synchronizace dat,
- datový model – vytvoření konkrétních komunikačních a datových objektů.

4.1.1 Komunikace

Aplikační server GlassFish nabízí podporu implementace REST rozhraní. Rozhraní je implementováno Java třídou s metodami a potřebnými anotacemi, interní zpracování požadavku již zřizuje aplikační framework. Třída musí mít před svojí deklarací definovanou tzv. *path*. Ta určuje, která třída a metoda má být volána po přijetí požadavku na konkrétní URL. Následující část znázorňuje deklaraci třídy a její metody pro zpracování požadavku na start *procesů*:

```
@Path("process-definitions")
public class RSProcessDefinitions {

        @POST
        @Path("start")
        @Consumes(MediaType.APPLICATION_JSON)
        @Produces(MediaType.APPLICATION_JSON)
        public Response startProcessDefinitions(StartProcess process){
        .
        .
        .
        }

}
```

Po přijetí požadavku na URL:
`http:/IP_SERVER:PORT/process-definitons/start`
je tento požadavek zpracován metodou *startProcessDefinitions* a následně je vrácena odpověď ve formátu JSON.

Tímto způsobem je implementováno celé REST rozhraní poskytující metody pro klientské žádosti na zpracování či synchronizaci dat.

4.1.2 Komunikační model

Samotné komunikační objekty a jejich atributy pro komunikaci mezi serverem a Activiti webovou službou byly určeny formátem JSON, který webová služba vrací (2.3). Podle struktury dat byly implementovány potřebné objekty nejprve pro komunikaci mezi serverem a Activiti webovou službou.

Příklad formátu JSON popsaný v kapitole 2.3 určuje, jak webová služba vrací konkrétní data například o objektu *proces*. Tato data jsou obsažena v poli s názvem *data*. Pro přijetí a zpracování celé odpovědi je potřeba přijmout objekt celý, včetně

doplňujících údajů (*total, start, sort*). Proto byly navrženy objekty, u kterých jsou jejich atributy shodné s atributy formátu JSON. Postup návrhu a implementace objektů jsou popsány v následujícím příkladu.

Po odeslání požadavku za účelem získání seznamu *procesů* z Activiti databáze je přijata odpověď ve formátu:

```
{
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1,
  "data": [
    {
      "id": "create_new_account:1",
      "key": "new_account",
      "version": 1
    }
  ]
}
```

Pro zpracování těchto dat jsou deklarovány následující třídy popisující požadované objekty (ukázka kódu je pouze ilustrační):

```
class ListProcessDefinitions{
      private int        total;
      private int        start;
      private String     id;
      private String     order;
      private int        size;
      private ArrayList<ProcessDefinition> data;
}

class ProcessDefinition{
      private String     id;
      private String     key;
      private int        id;
}
```

Výsledná, původně požadovaná data, jsou získána jako seznam objektů *ProcessDefinition* z přijatého objektu *ListProcessDefinitions*. Tímto způsobem je navržena a implementována také komunikace mezi serverem a klientem.

4.1.3 Autentizace klientů

Aktuální způsob autentizace klientů zprostředkovává server. Klienti jsou dle svého klientského ID a hesla ověřováni oproti webové službě Activiti *frameworku*. Po úspěšné autentizaci je zaručena následná možnost komunikace konkrétního uživatele s danou webovou službou. Každý další požadavek pro komunikaci s Activiti webovou službou musí obsahovat ve své hlavičce klientské ID a heslo.

Pokud má klient nastaven na straně serveru přístup k více datovým zdrojům (Activiti webových služeb), je při procesu přihlášení ověřen oproti všem databázím. Výsledek autentizace je nejen přeposlán klientovi, ale také uchován na straně serveru. Pokud autentizace selže (uživatel již nemá účet v databázi, chybné ID nebo heslo), synchronizace dat se s touto databází dále neprovádí.

Každý klient má na straně serveru minimálně jeden záznam v tabulce *user_source*. Při požadavku na autentizaci server z databáze získá všechny záznamy pro konkrétního uživatele a provede autentizaci oproti všem webovým službám.

Výsledek autentizace je uložen k odpovídajícímu záznamu jako atribut *isActive*. Tím je uchován stav poslední autentizace klienta, který je kontrolován při případné synchronizaci.

Po úspěšné autentizaci klienta si server vyžádá kompletní seznam skupin, ve kterých je konkrétní uživatel členem. Tyto informace jsou uloženy v databázi na straně serveru a jsou dále využity při datové synchronizaci pro získání všech dostupných *úkolů*.

4.1.4 Synchronizace dat

Kompletní logika aplikace byla při konečném návrhu aplikace přesunuta na stranu serveru. Klientská část proto nebude muset být ve většině případů změny modifikována, případné změny se budou týkat pouze centrálního prvku – serveru.

Server má za úkol nejen autentizaci klientů, ale také synchronizaci jejich dat. Výhodou centrální synchronizace je možnost modifikace a kontrola dat, jež jsou následně odeslány klientům.

V této práci byla implementována synchronizace dle konečného návrhu (3.3.2). V první fázi synchronizace jsou dle požadavku odpovídající data klienta na straně serveru sesynchronizována s Activiti databází, poté jsou synchronizována samotná data mezi aktuálními daty na serveru a koncovým klientem.

Klient vždy pouze odesílá žádost o synchronizaci (obsahující informaci o vlastních datech) a přijme synchronizační odpověď s již synchronními daty.

Jazyk Java nabízí práci s množinami pomocí kolekce *Set*. Při implementaci byla vytvořena třída *Synchronization*, jež svými metodami obaluje práci s množinamy a implementuje metody, jako jsou například vzájmené odečtení množin nebo jejich průnik.

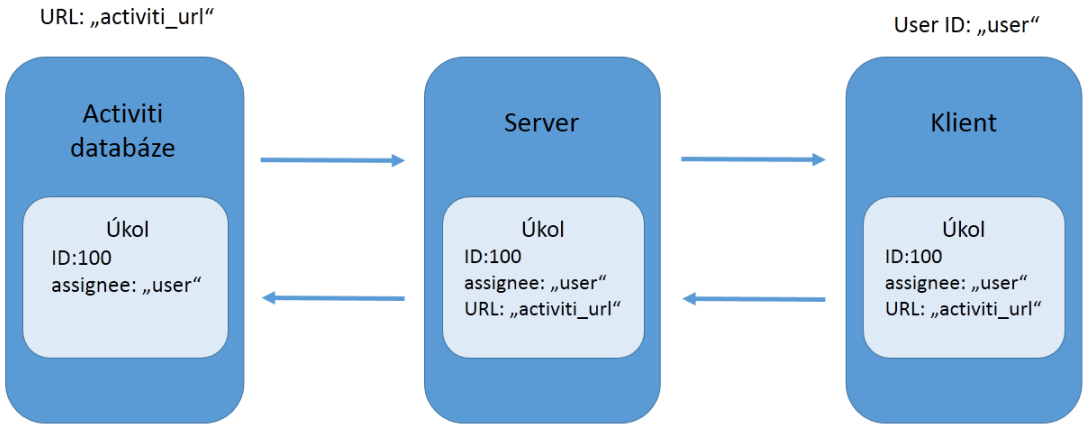
Výsledný rozdíl vrací opět jako množinu objektů, která je dále zpracována. K tomu, aby mohla být množina objektů odečtena od jiné, je potřeba, aby konkrétní objekt implementoval rozhraní *Comparable* a jeho metodu *compareTo*, ve které již vývojář musí implementovat kritérium vzájemného porovnání.

V prvotní implementaci byl, jako kritérium pro porovnání, zvolen složený identifikátor, který se skládal ze dvou částí. První z nich byl identifikátor dat získaný z Activiti databáze (ID *úkolu*), část druhou tvořilo URL Activiti webové služby, ze které data pochází. Tímto identifikátorem byla zaručena jednoznačnost každého datového objektu ve zpracovávané množině.

Během testování se ovšem tato implementace prokázala jako nedostačující z následujícího důvodu. Pro zjednodušení si lze představit příklad s jediným *úkolem* v Activiti databázi a jedním klientem (obrázek 11). Každý objekt *úkol* obsahuje několik atributů (například *description*), které se mohou na základě uživatelské potřeby změnit, tuto změnu je nutné distribuovat i mezi ostatní klienty.

Modelová situace vypadá následovně:

- Activiti databáze – obsahuje *úkol* s ID 100, tento *úkol* má zpracovat uživatel *user* (*user ID* je v poli assignee viz. 2.4).
- Server – již ve své databázi tento *úkol* uchovává, při jeho přijetí přidal k tomuto objektu hodnotu atributu *URL*, pro jeho jedinečnou identifikaci.
- Klient – jehož ID je *user* již také tento *úkol* uchovává ve své databázi.



Obrázek 11: Modelová situace problému synchronizace

Nyní dojde ke změně dat v Activiti databázi (například description). Změní se pouze description, ID tohoto *úkolu* zůstane nezměněno. Následně klient zažádá o synchronizaci. Server si vyžádá všechny *úkoly* pro uživatele *user*. Obdrží pouze jediný s identifikátorem 100 a s URL *activiti_url*. Tato data sesynchronizuje s daty z vlastní

databáze – opět *úkoly* pro uživatele *user* s ID 100 a URL *activiti_url*. Porovnávací klíč je složen právě z hodnot 100 (ID *úkolu*) a *activiti_url*(URL webové služby), proto je synchronizace vyhodnocena s výsledkem *data aktuální*. Klient je tedy informován, že všechna jeho data jsou v aktuální podobě.

Tento výsledek je ovšem nepravdivý, protože původně došlo ke změně *description*. Tato změna nebyla zohledněna během synchronizace, a proto bylo nutné navrhnout a implementovat sofistikovanější řešení daného problému, které by zohledňovalo případnou změnu dat v databázi (nejen ID *úkolu*).

V případě shodných ID *úkolů* bylo možné postupné porovnávání všech vzájemných atributů objektů (například *object1.description* vs *object2.description* atd). Tento postup by byl ovšem velmi neefektivní především z časové a výpočetní náročnosti. Proto bylo implementováno následující řešení.

Server při přijetí *úkolu* z Activiti databáze k tomuto objektu doplní nejen *activiti_url*, ale také vypočítá HASH z požadovaných atributů, u kterých mohlo dojít ke změně. Při synchronizaci již není porovnání prováděno na základě ID a URL, ale na základě vypočítané hodnoty HASH. Případná shoda HASH značí identický úkol včetně všech atributů, případná neshoda značí změnu některého atributu.

V jazyce Java bylo tohoto řešení docíleno využitím anotací u požadovaných atributů. Použití vlastního anotační rozhraní *TaskHash* a *java.lang.reflect* je umožněno získat hodnoty atributů objektu označené konkrétní anotací. Z těchto hodnot po přijetí dat na serveru vypočítat hodnotu HASH a uložit ji jako atribut objektu.

Výsledná třída UserTask je tedy definována s implementací *Comparable* rozhraní a jednotlivé atributy, u kterých může dojít ke změně, jsou označeny anotací *TaskHash*:

```
public class UserTask implements Comparable<UserTask> {

        @TaskHash
        private int id;
        @TaskHash
        private String name;
        .
        .
}
```

Z takto označených atributů je vypočítán HASH, který je následně porovnáván. Implementace porovnání řetězců v jazyce Java je doporučována pomocí metody *equals* (dle Java tutorial [22]). Tento způsob se během následného testu ukázal jako nedostačující. Některé HASH hodnoty, i přesto, že se shodovaly, tato metoda vyhodnotila jako různé a synchronizace nezaručovala korektnost dat.

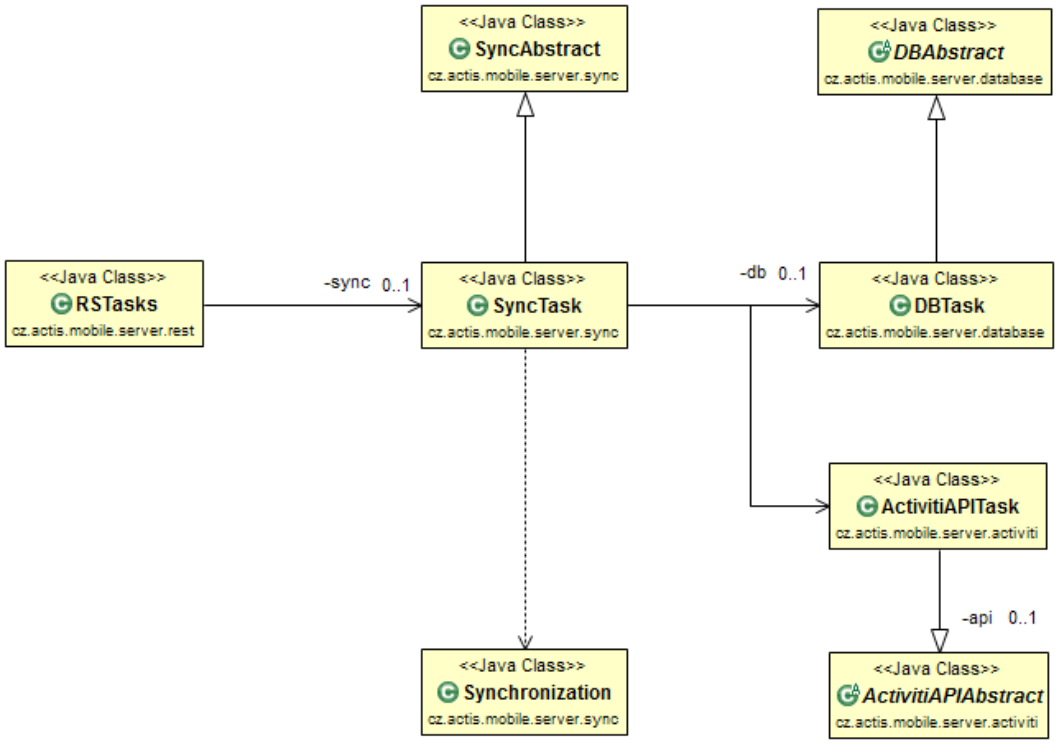
Tento problém je řešen využitím třídy *Collator*, která slouží k lingvisticky správnému porovnání textových řetězců. Tato implementace se během testování prokázala jako dostatečně rychlá a ve všech testovaných případech poskytovala korektní výsledek porovnání.

Na UML diagramu číslo 12 je znázorněn vztah mezi třídami implementujícími příjem požadavku, zpracování a odeslání odpovědi. Schéma zobrazuje třídy pro zpracování požadavku, který má sesynchronizovat uživatelské *úkoly*.

Aplikace byla vyvinuta pro snadné rozšíření stávající funkcionality. Proto jsou třídy implementující zpracování konkrétních objektů rozšířeny od tříd abstraktních, které již implementují společné metody pro práci s datovými objekty, přístup do databáze, využití logování či například komunikaci se samotnou webovou službou poskytující Activiti *framework*.

Při případném rošíření aplikace o REST rozhraní a například zpracování nového požadavku je potřeba pouze rozšířit již připravené abstraktní třídy a implementovat konkrétní metody pro zpracování požadavku.

Na straně serveru nebyl využit *Entity framework*, ale vytvořena vlastní vrstva mapující data z databáze do podoby příslušných objektů. Veškeré práce s daty jsou prováděny pro zaručení konzistentnosti dat pomocí transakcí.



Obrázek 12: UML schéma tříd pro zpracování synchronizačního požadavku

4.2 Klient

Tato kapitola popisuje implementaci klientské části aplikace a uvádí hlavní prvky implementačních rozdílů mezi vývojem desktopové a mobilní klientské aplikace.

Cílem práce bylo implementovat klientskou část primárně pro platformu Android [6]. Pro možnost efektivnějšího ladění aplikace a přístupu do databáze (interního v rámci firmy), se ukázalo nejvhodnějším řešením vytvořit nejprve desktopovou klientskou aplikaci v jazyce Java s vlastní MySQL databází. Tato klientská část slouží především pro testování komunikace a pro demonstrační účely v rámci interního využívání aplikace a byla vytvořena nad rámec původního zadání práce. Až po schválení požadované implementace a funkčnosti desktopové aplikace byla vyvíjena klientská mobilní aplikace pro platformu Android.

Klientská aplikace byla od počátku navržena jako jednouživatelská s intuitivním ovládáním. Důvodem byl především předpoklad práce s mobilním zařízením, které je vlastněno jednou osobou.

4.2.1 Bezpečnost interních dat

U platformy Android byl zadavatelem vznesen požadavek na zabezpečení lokálních dat v mobilním zařízení. Platforma Android nabízí jako interní úložiště nezabezpečenou SQLite databázi. Pří ztrátě mobilního zařízení nebo instalaci *škodlivé* aplikace (například z internetu) je bezpečnostním problémem možnost získat nezabezpečená data z interní SQLite databáze. Z tohoto důvodu bylo implementováno zabezpečení databáze, kterou zprostředkovává knihovna SQLCipher [25].

Snahou vývojářů této knihovny bylo zvýšení bezpečnosti a soukromí svých uživatelů. SQLCipher je rozšíření klasické SQLite knihovny, které poskytuje 256-bitové AES šifrování. Nespornou výhodou této knihovny je shodné API, které poskytuje pro přístup k databázi jako klasické SQLite.

Během vývoje aplikace se nejprve pracovalo s nezabezpečenou databází. Po návrhu databáze (obrázek 7) a otestování korektnosti návrhu, které proběhlo na dříve vytvořené desktopové aplikaci, bylo nutné doimplementovat knihovnu SQLCipher.

Při rozšíření projektu o zabezpečení lokálních dat u mobilního klienta byl využit postup společnosti *Zatetic*, který popisuje integraci SQLite knihovny do aplikace pro Android [26]. Prokázalo se, že vývojáři skutečně vytvořili shodné rozhraní, které poskytovala původní SQLite knihovna. Téměř jediným rozdílem je nutnost zadat heslo při práci s databází a při startu aplikace nahrát do paměti potřebné knihovny realizující šifrování a dešifrování dat. Implementace zabezpečení ovšem přinesla nárůst velikosti výsledného apk instalačního souboru z původních 1,1MB na 5,1MB. Dále jsou při prvním spuštění aplikace rozbaleny na disk potřebné knihovny pro šifrování. Výsledná aplikace tím zabírá po instalaci a spuštění až 15MB. Proto je umožněn její

přesun na SD kartu mobilního zařízení (v případě podpory verze systému Android). SD karty v dnešní době disponují velikostí paměti i několik desítek GB.

4.2.2 Komunikace

Obě klientské aplikace implementují komunikaci přes REST rozhraní. Komunikační model zůstává u obou aplikací stejný, ovšem konkrétní implementace se v některých částech liší. U desktopové části je komunikace implementována ve více třídách a metodách. Účelem tohoto postupu je snadná možnost modifikace způsobu odesílání dat, příjmu dat a změny potřebných parametrů pro potřeby ladění a testování. Vývoj komunikačního rozhraní mobilní aplikace byl realizován již oproti otestovanému rozhraní implementovaného serveru. Proto bylo možné navrhnout a implementovat obecnější rozhraní pro vzájemnou komunikaci než u desktopového klienta. Nebylo zde nutné kontrolovat přijatá data ze serveru.

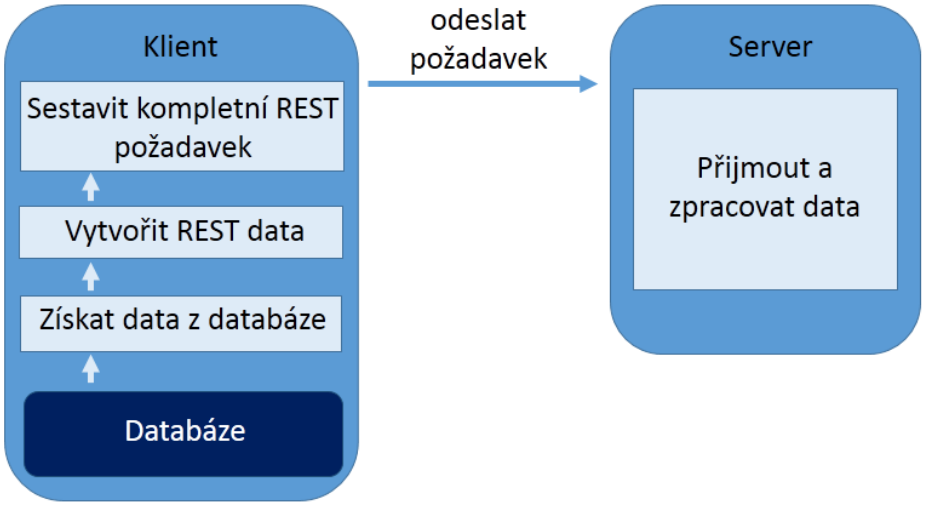
Desktopová klientská část využívá *Jersey framework* [17] pro implementaci REST komunikace se serverem. Tento nástroj vývojářům výrazně usnadňuje implementaci komunikace, odeslání konkrétních objektů a zároveň zpětné mapování příchozích dat na příslušné objekty. Další výhodou je přímá podpora a využití Jersey frameworku také na straně serveru. To umožnilo použít shodné API na obou komunikačních stranách.

V průběhu vývoje se projevil i drobný nedostatek Jersey frameworku. Jersey framework při vytváření komunikačního objektu ve formátu JSON chybně mapuje jednoprvkové pole, které převádí na klasický atribut objektu. Přenos dat je proveden korektně, avšak při zpětném složení objektu je vrácena v některých případech výjimka. Jersey metoda očekává podle předpisu třídy atribut pole (uvozený v hranatých závorkách) a získá pod tímto jménem *pouze* jednoprvkový atribut (uvozený v klasických závorkách). Tento problém byl vyřešen využitím knihovny *Jackson Object Mapper* [1]. Jedná se o část frameworku pro práci s JSON objekty, který korektně převádí Java objekty do formátu JSON a zpět.

Součástí kouminkace za účelem synchronizace nejsou pouze `kompletní` objekty (*úkol, proces*). Při odeslání požadavku na synchronizaci klient předává také informace o tom, jaká data vlastní ve své databázi. V tuto chvíli není nutné přenášet `kompletní` objekt, ale stačí pouze identifikátor sloužící pro synchronizační proces na serveru. V kontrétním případě pro synchronizaci *úkolů* klient odesílá ve své zprávě pouze *taskID, activitiURL a HASH*. Průměrná znaková délka objektu *úkol* ve formátu JSON se pohybuje mezi 600–800 znaků (záleží na délce popisu, URL atd). Synchronizační REST požadavek (obsahující *taskID, activitiURL a HASH*) má v průměrném případě délku pouze 150 znaků, což je maximálně 1/4 původního objektu. Tím je minimalizován datový přenos a zvýšena rychlost aplikace.

Schéma 13 popisuje obecné vytvoření požadavku na synchronizaci, odeslání a příjem na straně serveru. Klient nejprve složí požadavek z dat uložených v da-

tabázi. Pojmem *REST data* rozumíme vytvoření objektů určených k synchronizaci (vytvoření *REST objektů s minimálním počtem potřebných atributů*). Posledním krokem před odesláním je složení celého požadavku včetně dat o samotném uživateli, který o synchronizaci žádá (například klientský identifikátor a heslo). Následně je požadavek odeslán na stranu serveru.



Obrázek 13: Schéma vytvoření synchronizačního požadavku

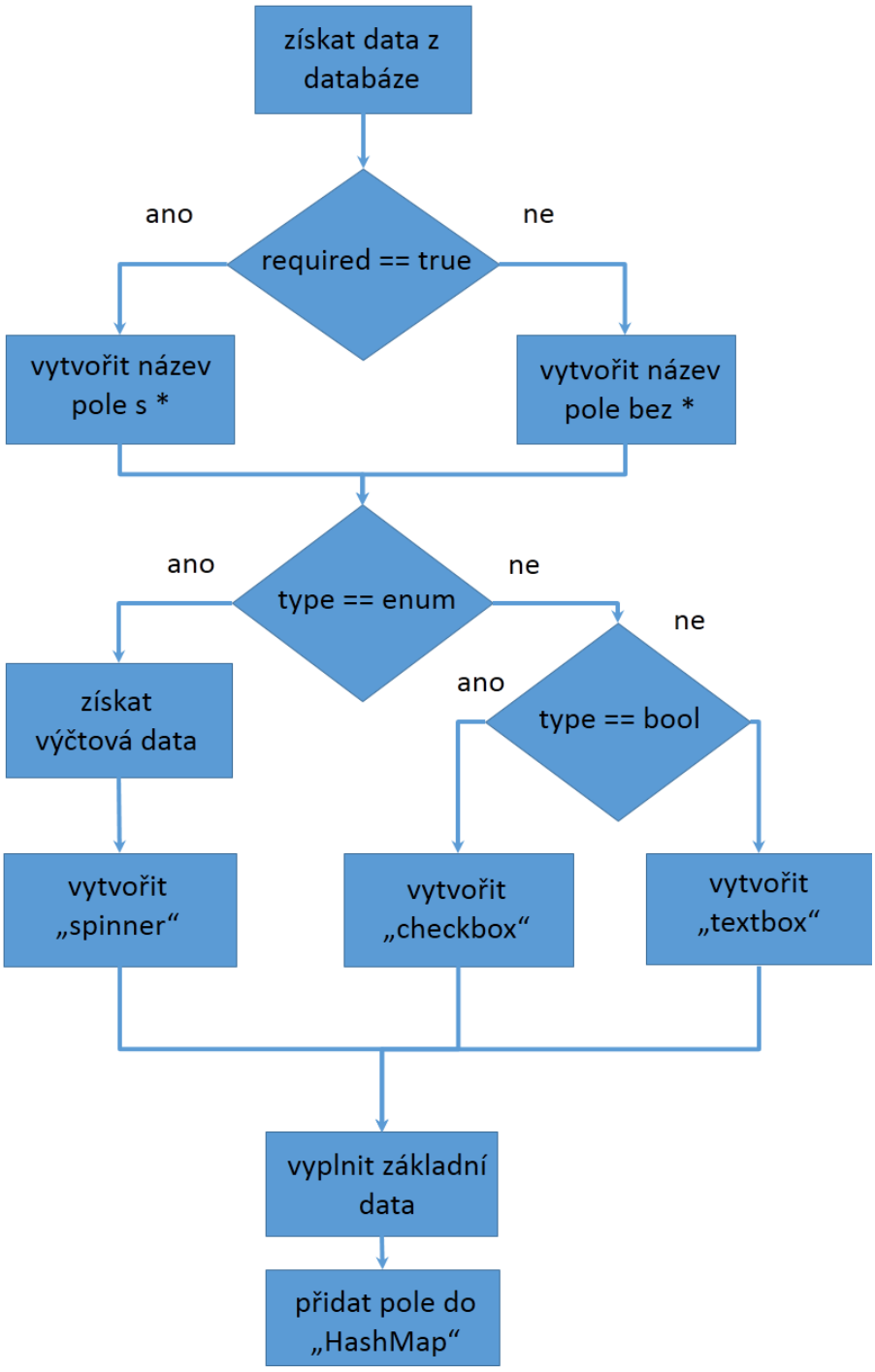
4.2.3 Generovaní formulářů

Jak je popsáno v kapitole 2.4, interakce s uživateli je zajištěna pomocí formulářů. V klientské aplikaci je potřeba z přijatých dat dynamicky sestavit a zobrazit formulář ke konkrétnímu *úkolu*. Uživatelem vyplněná data zvalidovat a uložit zpět do databáze. Uložená data se při následné synchronizaci opět připojí k ukončovanému objektu *úkol* a jsou společně odeslána na server.

Generování formuláře se skládá z několika kroků. Aplikace umožňuje validovat formulář obsahující pole s datovými typy *string, long, enum, boolean*. Každé pole je představováno samostatným objektem, který má například vlastní ID, typ, název, předvyplněnou hodnotu a příznak, zda je vyplnění pole povinné.

Pokud není pole typu *enum* ani *boolean*, je vygenerován klasický *editText* pro zadání dat. Jako jméno pole je využit atribut název a pokud je pole povinné, je za název přidána hvězdička. Při zobrazení typu *enum* jsou z databáze získána dopňující data, která jsou uživateli zobrazena jako konečný seznam, ze kterého si uživatel volí konkrétní hodnotu. Postup vytvoření jednoho pole formuláře je zobrazen pomocí diagramu 14. Jako datová struktura pro uchování dynamického stavu polí formulářů je v jazyce Java využita *HashMap* s identifikátorem pole a jeho instancí. Typ pole *boolean* je reprezentován *checkboxem*.

Před uložením dat z formuláře jsou data zvalidována dle typu pole. Pokud je atribut pole například *long* a vstupní data nelze přetypovat na tento typ, je uživatel informován o chybném vstupu.

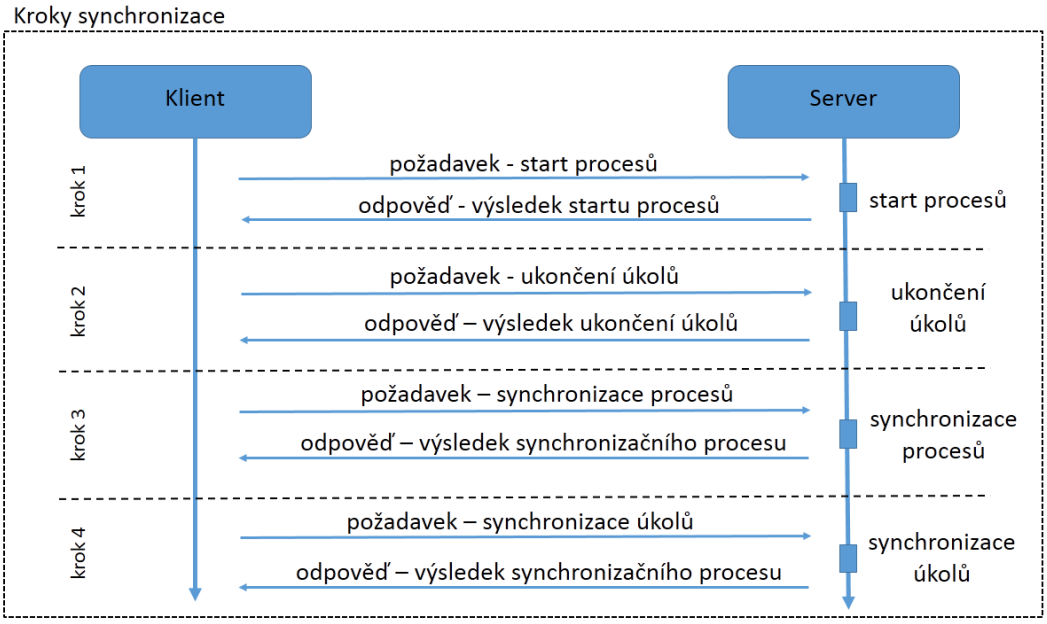


Obrázek 14: Diagram vytvoření formulářového pole

4.2.4 Synchronizace klientské části

Synchronizační mechanismus na straně serveru je popsán v kapitole 3.3.2. Klientská část má tedy za úkol pouze odesílat požadavky a zpracovat příchozí data v odpovědi.

Synchronizační postup se skládá celkem ze 4 kroků, přitom první 2 kroky není potřeba vykonávat při každé synchronizaci a záleží na aktuálním stavu dat v klientské aplikaci. Postup synchronizace je znázorněný na následujícím schématu.



Obrázek 15: Kroky synchronizace

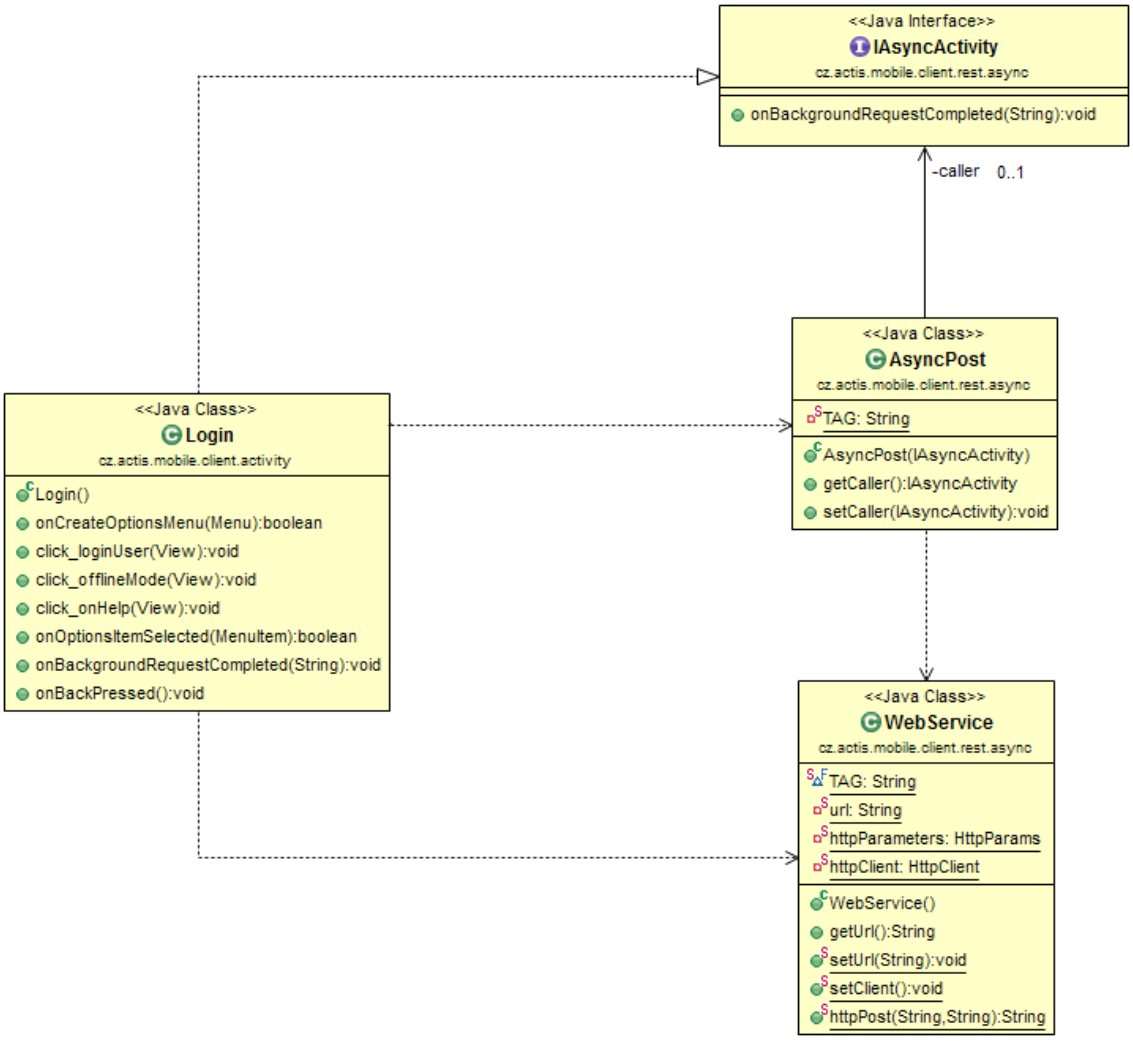
Synchronizace se pro uživatele aplikace jeví jako jednotná operace, ve které jsou vykonány postupně následující kroky (přitom je v každém kroku nejprve složen požadavek dle schématu 13):

- start procesů – požadavek je vytvořen a odeslán, pokud uživatel zvolil start *procesu* po poslední synchronizaci. Pokud uživatel nezažádal o start žádného *procesu*, není nutné požadavek odesílat,
- dokončení *úkolů* – požadavek je vytvořen a odeslán, pokud uživatel v klientské aplikaci po poslední synchronizaci dokončil některý *úkol*, pokud se tak nestalo, tento požadavek také není nutné odesílat,
- synchronizace procesů – tento požadavek je odesílán při každém procesu synchronizace, složený požadavek obsahuje identifikátor procesů, které má klient ve své databázi,
- synchronizace *úkolů* – požadavek je také odeslán při každé synchronizaci, opět jsou jeho součástí identifikátory *úkolů*, které má již klient uložené ve své lokální databázi.

4.2.5 Implementační rozdíly klientských částí

V některých částech implementace se desktopová klientská část liší od implementace aplikace pro platformu Android. Postupováno bylo podle praktických příkladů pana Larse Vogela [23]. Implementace komunikace klientské části pro platformu Android je realizována pomocí asynchronního vlákna, ve kterém je odeslán požadavek. Třída, ve které je požadavek odesílán, implementuje rozhraní *IAsyncActivity*. Tím je zaručena obsluha společné metody *onBackgroundRequestComplete*, která je volána po přijetí odpovědi a ukončení činnosti asynchronního vlákna.

Příklad implementace autentizace klienta pro platformu Android je znázorněna UML diagramem číslo 16.



Obrázek 16: UML schéma tříd pro odeslání autentizačního požadavku

Znázorněná třída *Login*, která zároveň obsluhuje přihlašovací formulář, složí požadavek (zadané uživatelské ID a heslo), nastaví třídě *WebService* URL, na které je následný požadavek odesílán. Odeslání je realizováno metodou *execute*, kterou je vytvořeno nové vlákno a třídou *AsyncPost* je obslouženo odeslání požadavku. Po přijetí

odpovědi je díky implementaci společného rozhraní zavolána právě metoda *onBackgroundRequestComplete* a předána odpověď. Ta je poté zpracována a uživateli je na základě výsledku autentizace případně povolen vstup do aplikace.

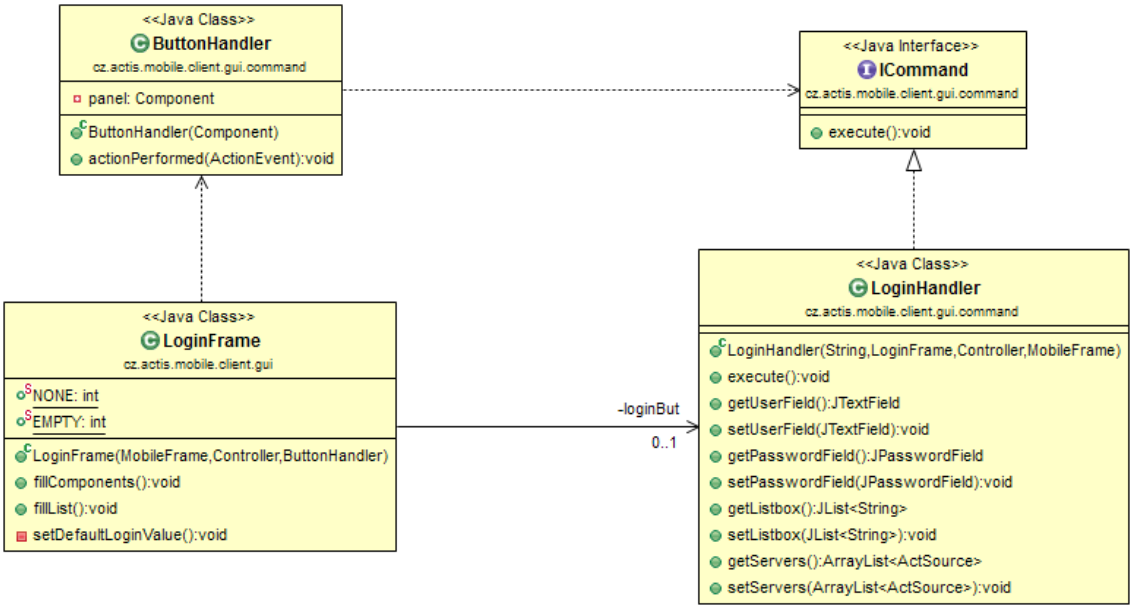
Tímto způsobem je přes jednotné univerzální rozhraní implementováno i odesílání všech synchronizačních požadavků.

V klientské desktopové části aplikace je synchronizace prováděna stejnými kroky, pouze pro tento proces není vytvářeno samostatné vlákno a metody pro odesílání či příjem lze libovolně konfigurovat pro potřeby testování. Odesílaná a přijímaná data klientskou aplikací jsou kompletně logována pro případné ladění chyby při přenosu dat.

Dalším rozdílem ve vývoji je celkový přístup k návrhu výsledné klientské aplikace. Zatímco u desktopové klientské části je umožněno data mezi objekty předávat pomocí konstruktoru, tento přístup ve vývoji pro platformu Android použít nelze. Pokud chceme předat nějaké activitě (třídě obsluhující právě jedno zobrazení na mobilním zařízení) objekt, je nutné, aby třída implementovala *Parcalable* rozhraní a jeho metody. Teprve poté je umožněno předání instance této třídy (objektu) jiné aktivitě při jejím startu.

4.2.6 Využití návrhového vzoru Command u desktopové aplikace

Při vývoji aplikace byly využity některé z návrhových vzorů nebo jejich částí. U desktopové klientské aplikace byl například využit návrhový vzor *Command*. Tento vzor je využíván k obsluze stisknutí tlačítka uživatelem.



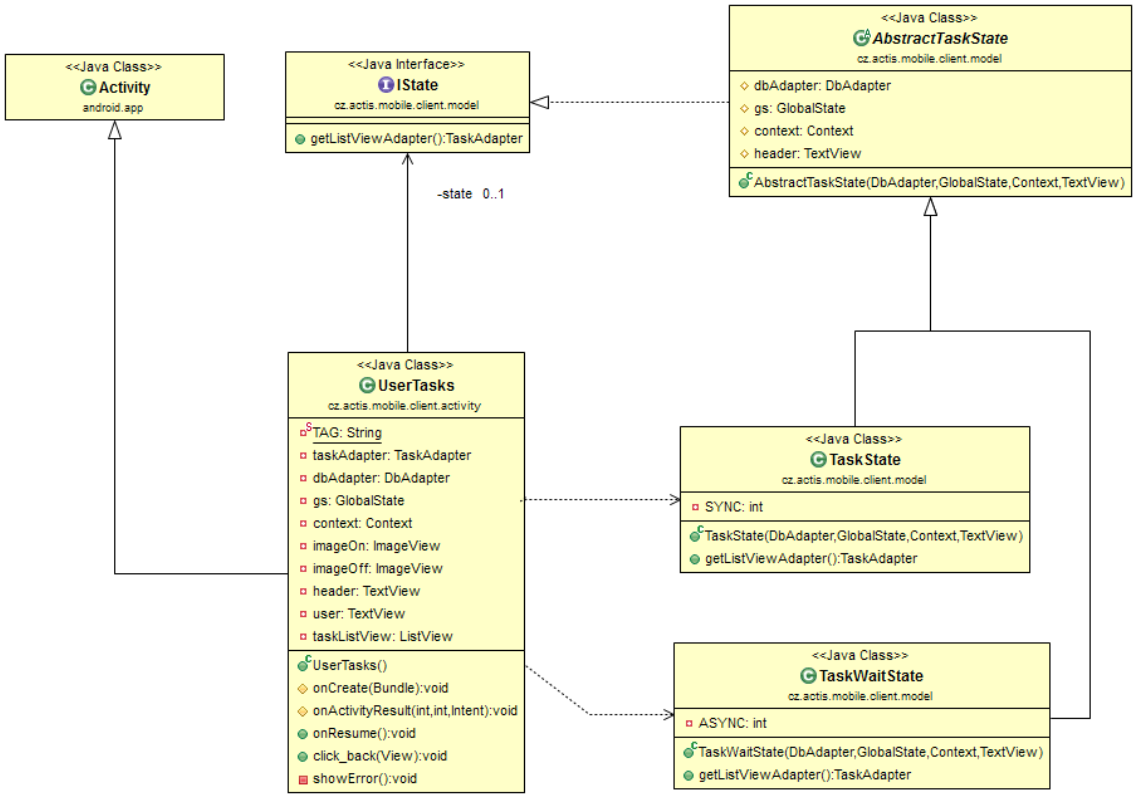
Obrázek 17: UML využití návrhového vzoru Command

UML diagram 17 zachycuje vztahy mezi třídami po implementaci *Command*

návrhového vzoru. Vlastní třída *ButtonHandler* slouží k obsluze metody *execute*, kterou díky společnému rozhraní *ICommand* definuje každá třída představující konkrétní tlačítko v grafickém uživatelském rozhraní.

4.2.7 Využití návrhového vzoru State u mobilní aplikace

Při implementaci mobilní aplikace se ukázalo vhodné použít návrhový vzor *State*. Aktivita pro zobrazení seznamu nezpracovaných *úkolů* a aktivita pro zobrazení seznamu *úkolů* čekajících na synchronizaci se liší pouze v popisku a zobrazovaných datech. Proto na obě aktivity stačí pouze jedna třída, která popisek a cílová data zobrazuje v závislosti na vnitřním stavu (*IState*). Vztah mezi třídami je znázorněn na následujícím UML diagramu.



Obrázek 18: UML využití návrhového vzoru State

4.2.8 Mobilní aplikace – back stack

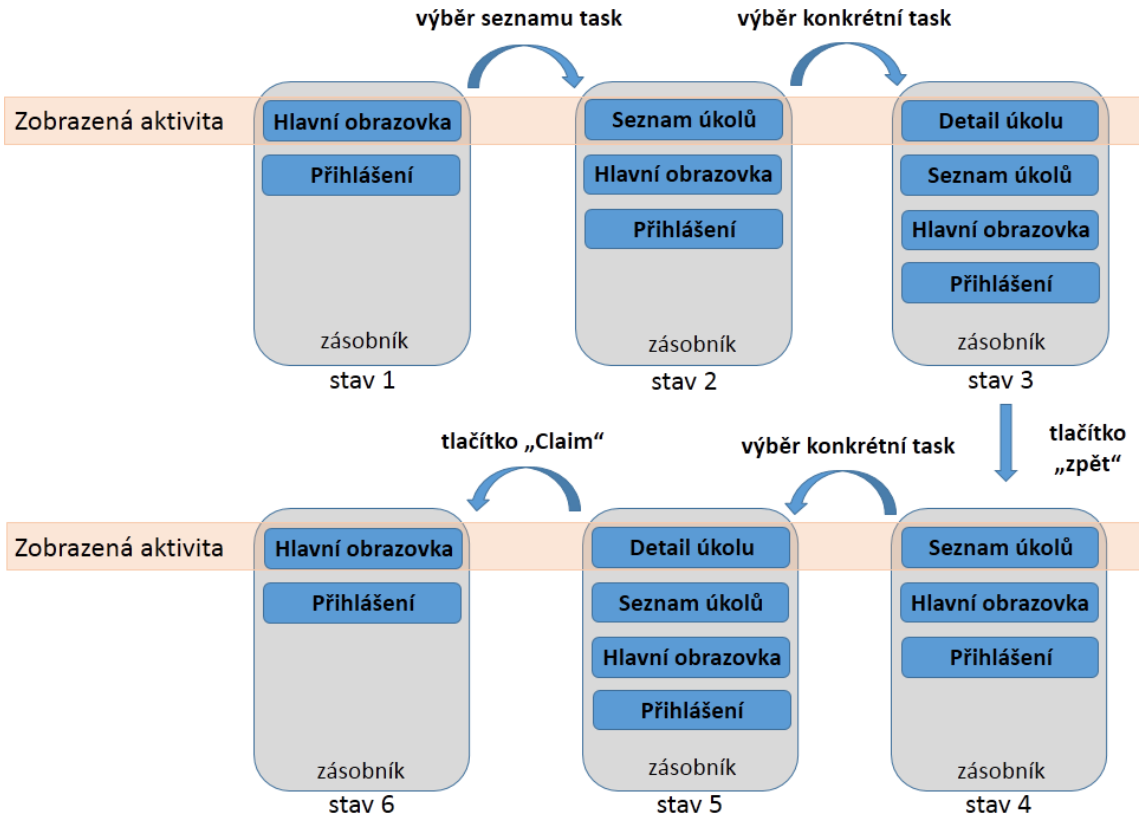
Platforma Android poskytuje pro práci s activitami tzv. *back stack* [6] neboli zásobník aktivit. Umístění activity na vrcholu zásobníku je řízeno podle životního cyklu každé aktivity [9].

Při spuštění aplikace je spuštěna tzv. *MAIN* aktivita, konkrétně přihlašovací formulář. Po přihlášení je nastartována aktivita se základním rozcestníkem aplikace – *main screen*. Pokud přecházíme mezi dalšími aktivitami (dokončení *úkolů*, start *pro-*

cesů), je vždy minulá aktivita přesunuta do takzvaného back stacku, nová přejde automaticky do popředí a je zobrazena. Při kliknutí na tlačítko *back* je aktuální aktivita zrušena a zobrazí se poslední aktivita uložená na vrcholu *back stacku*.

Pokud uživatel dokončí *úkol* (*complete, claim, unclaim*), není potřeba se vracet na předchozí aktivitu, ve které jsou zobrazeny pouze detaily aktuálně dokončeného *úkolu*. Z uživatelského hlediska je flexibilnější zobrazit po dokončení *úkolu* opět základní rozcestník aplikace – *main screen*.

Tohoto je docíleno před ukončením jedné aktivity nastavením tzv. *result*. Tento argument je předán aktivitě, jež je na vrcholu *back stacku* a přechází do popředí. V této aktivitě lze předaný argument zpracovat a zvolit vhodnou reakci. V tomto případě například ukončit i aktuální aktivitu.



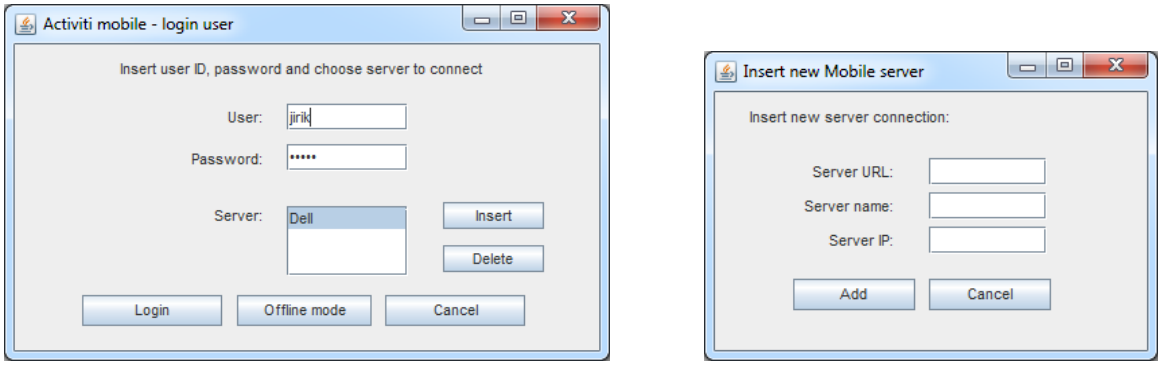
Obrázek 19: Schéma práce s aktivity zásobníkem [7]

Znázornění funkce práce se zásobníkem je na schématu číslo 19. Po přihlášení uživatele (stav 1) je aktivní aktivita *Hlavní obrazovka* a v zásobníku (*back stacku*) je aktivita předchozí (*Přihlášení*). Uživatel zvolí zobrazení seznamu *úkolů*, tím je nastartována nová aktivita, která je zobrazena uživateli (stav 2). Ze seznamu uživatel zvolí konkrétní *úkol* a nastartuje další aktivitu (stav 3). Poté klikne na tlačítko zpět, aktuální aktivita je zrušena a aktivní se stává předchozí *Seznam úkolů* (stav 4). Znovu zvolí konkrétní *úkol* (stav 5) a poté tento *úkol* dokončí s rozhodnutím *Claim*. Nyní je

ovšem na rozdíl od funkce tlačítka zpět předán aktivitě *Seznam úkolů* argument, který je vyhodnocen jako ukončující, a i tato aktivita je zrušena (stav 6). Z *Hlavní obrazovky* lze přejít tlačítkem zpět vždy jen na přihlašovací obrazovku.

4.2.9 Možnosti desktopové aplikace

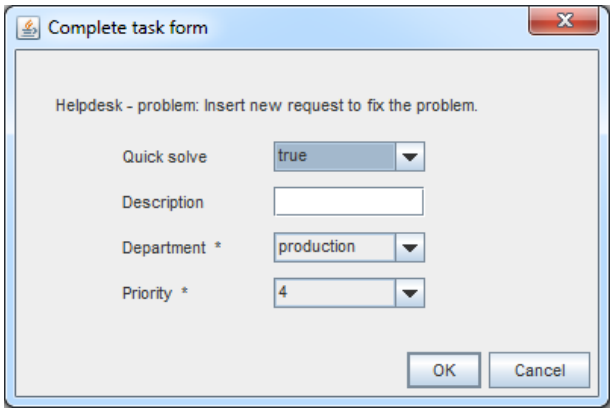
Jak již bylo zmíněno výše, desktopová aplikace slouží především pro interní testování přenosu dat a synchronizačního mechanizmu v rámci firmy. Byl proto implementován jednoduchý uživatelský design a ovládání aplikace. Uživateli je při přihlašování zobrazen formulář, ve kterém je možné vložit přihlašovací údaje a zvolit mód (on–line nebo off–line), ve kterém má být aplikace využívána. Také je možné smazat nebo přidat nové informace o serveru, ke kterému se má klient následně připojit.



(a) Přihlašovací obrazovka (b) Formulář pro přidání serveru

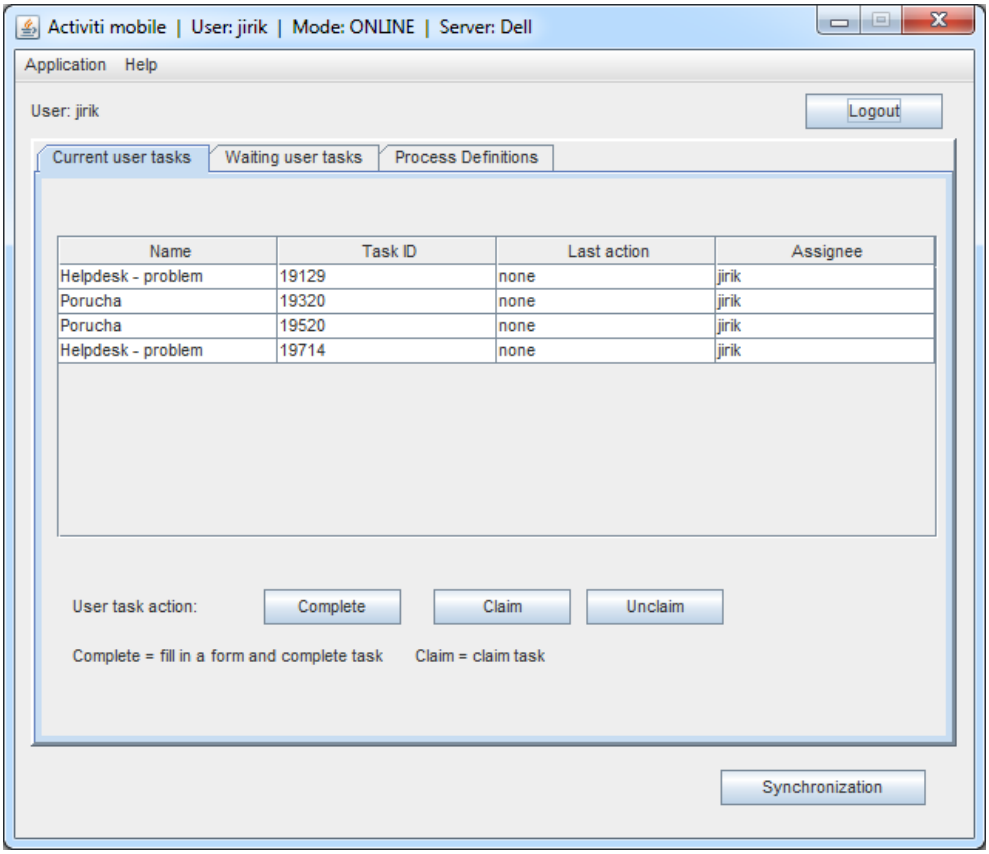
Obrázek 20: Přihlášení uživatele a konfigurace připojení

Důležitou součástí obou klientských částí je dynamické vygenerování formuláře pro dokončení *úkolu*. Uživatel vyplní formulář dle požadavků a následně jsou vyplněná data zvalidována. V případě korektního formátu vstupních dat jsou uložena do klientské databáze.



Obrázek 21: Dynamicky vygenerovaný formulář

Dále je práce s aplikací vymezena na jedno okno, ve kterém lze pomocí přepínání záložek měnit, zobrazovat a zpracovávat požadovaná data. Záložky jsou celkem tři. Na první z nich jsou v tabulce zobrazeny aktuální *úkoly*. Ve druhé záložce jsou zobrazeny *úkoly*, které již byly ukončeny a čekají na synchronizaci. V této záložce lze rozhodnutí o ukončení *úkolu* ještě před synchronizací změnit. Nakonec jsou pod třetí záložkou zobrazeny *procesy* včetně doplňujících informací.



Obrázek 22: Hlavní okno pro práci s daty

Pro přehledné testování a optimalizování aplikace jsou logovány téměř veškeré práce s daty. Jedná se především o práci s databází, složení a odeslání synchronizačních požadavků a v neposlední řadě výpis dat přijatých od serveru. Konfiguraci připojení k databázi a například úroveň logování lze modifikovat v tzv. *properties* souboru.

4.2.10 Možnosti mobilní aplikace

Mobilní klientská aplikace je dostupná v instalačním *apk* souboru pro platformu Android. Po instalaci je umožněn ve verzi systému, který to umožňuje, přesun aplikace na SD kartu.

Aplikace je implementována jako jednouživatelská, kde je uživatelské ID považováno za statický identifikátor. Proto je po prvním úspěšném přihlášení aplikace interně registrována na konkrétní uživatelské ID a jiný uživatel již nemá možnost

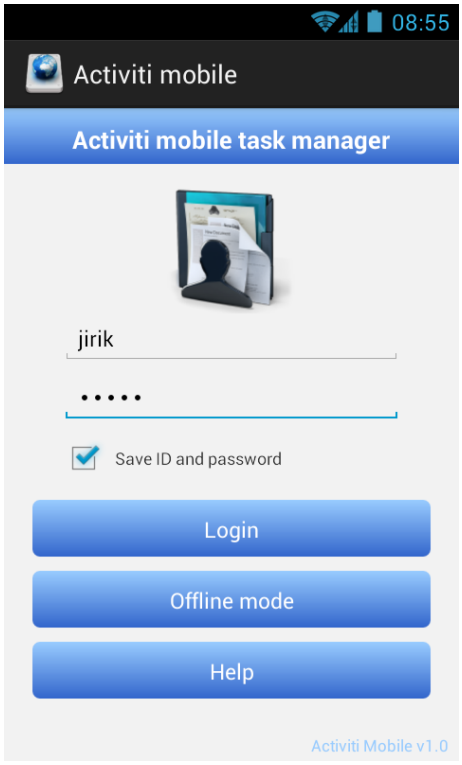
aplikaci využít. Pokud by ovšem došlo ke změně ID, musela by být aplikace kompletně odinstalována, znovu nainstalována a registrována na nové ID. Proto je v aplikaci možnost smazat nejen všechna interně uložená data, ale také celý tzv. *profil* uživatele. Poté je možné aplikaci využít znovu, jako by byla právě nainstalována.

Použití tzv. off–line režimu je uživateli umožněno až po úspešném prvním přihlášení na server. Do té doby se v interní databázi nemohou vyskytovat žádná data, která by uživatel mohl zpracovávat. Ta jsou získána až po první synchronizaci se serverem. Dále jsou přihlašovací data po prvním úspešném přihlášení uložena do databáze a jsou využita při přihlašování do tzv. off–line režimu. V základním nastavení aplikace jsou vyžadovány přihlašovací údaje nejen pro on–line režim, ale také pro off–line režim. Toto ověření lze v nastavení aplikace deaktivovat. Uložené přihlašovací údaje (uživatelské ID a heslo) je také možné při následnem spuštění automaticky předvyplnit pro rychlejší obsluhu aplikace.

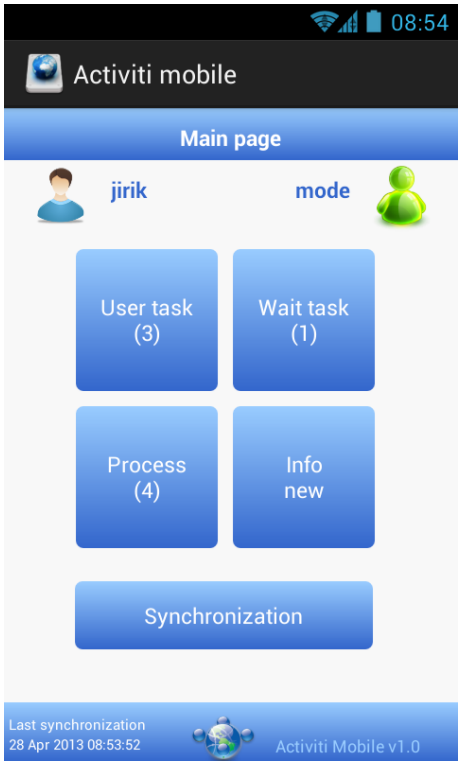
Data uložená v databázi mobilního zařízení jsou šifrována pomocí SQLCipher knihovny. Dále byly implementovány *pasivní* prvky zabezpečení, jako například odstranění aplikace z historie spuštěných aplikací. Není tedy možné přijít k mobilnímu zařízení a z historie spuštěných aplikací spustit tuto aplikaci. Systém Android obecně umožňuje při spuštění aplikace vrácení její poslední aktivity. To by znamenalo přímý přístup do zabezpečené části aplikace bez nutnosti přihlášení. Proto je po startu aplikace vždy zobrazen pouze přihlašovací formulář, ať už byla aplikace ukončena v jakékoliv aktivitě (například tlačítkem *home*).

Off–line režim umožňuje spravovat interně uložená data bez možnosti synchronizace se serverem. Dokončené *úkoly* v off–line režimu jsou ukládány do interní databáze. Všechna data jsou poté hromadně sesynchronizována při následné synchronizaci se serverem v on–line režimu. V off–line režimu je dále místo ikony zeleného panáčka indikujícího on–line připojení, zobrazena ikona panáčka červeného.

Aplikace se připojuje na server, proto je potřebná modifikace konfigurace připojení. Základní nastavení pro připojení k serveru je nastaveno testovací URL a jeho název. Pokud bude serverová aplikace spuštěna na jiném serveru, lze před přihlášením URL serveru modifikovat. Při případné ztrátě spojení se serverem je uživatel informován o chybě a aplikace je automaticky přepnuta do off–line režimu. Dále je na obrazovce hlavního rozcestníku aplikace zobrazeno datum a čas poslední synchronizace se serverem. Tím má uživatel zaručen přehled o aktuálnosti uložených dat.



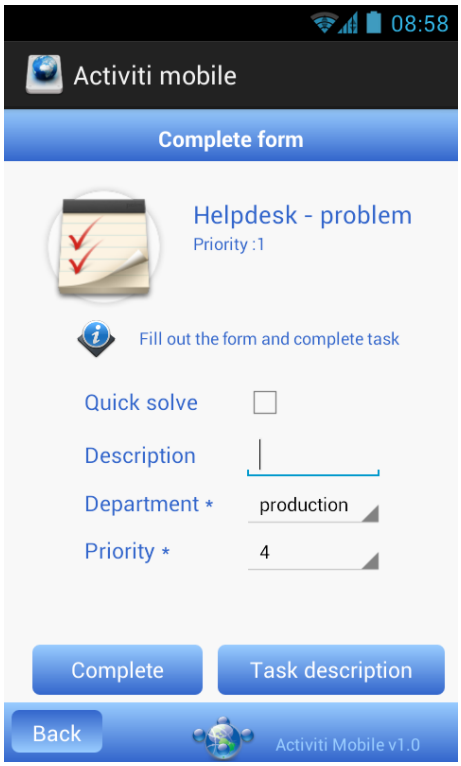
(a) Přihlašovací obrazovka



(b) Hlavní rozcestník aplikace

Obrázek 23: Přihlášení uživatele a konfigurace připojení

(a) Seznam dostupných procesů



(b) Dynamicky vygenerovaný formulář

Obrázek 24: Seznam procesů a zobrazení formuláře

Uživatelské rozhraní bylo, stejně jako u desktopové klientské aplikace, navrženo s důrazem na intuitivní orientaci v aplikaci s cílem snadného a rychlého ovládání. Proto je například v aplikaci tlačítko *zpět (back)* umístěno v levém dolním rohu, kde je jeho ovládání palcem nejpřirozenější při klasickém držení mobilního zařízení.

Jedním z cílů práce bylo také doplnit výslednou mobilní aplikaci o uživatelský manuál. Manuál mobilní aplikace je v příloze 7. V samotné aplikaci je kompletní manuál dostupný ve formě statické webové stránky jak z přihlašovacího formuláře, tak z hlavního zobrazení aplikace.

Uživatelská aplikace byla vyvinuta pro verzi systému Android 2.2+. Tato verze spolu s vyššími jsou dnes dle Android statistik [10] využívány ve většině mobilních zařízeních se systémem Android. Aplikace by proto mohla být široce využívána.

5 Testování aplikace

Výsledná aplikace byla v závěru otestována na aplikačním serveru GlassFish 3.1.2 s nastavením komunikace uživatele s dvěma Activiti *frameworky*. Pro simulaci reálného prostředí byla při testování zprovozněna webová služba nejen na lokálním stroji, ale také na serveru Microsoft Windows Server 2003. Testování bylo rozděleno na dvě části. První z nich bylo testování datového přenosu a korektní synchronizace na straně serveru. Následovala druhá týkající se zpracování dat v klientské části aplikace a komunikace s uživatelem. Během první fáze testování byl kladen důraz především na korektní:

- přenos dat – sestavení a opětovné kompletní složení datových objektů,
- komunikace s databází – ověření správné funkcionality především při práci s daty na straně serveru,
- synchronizační proces – porovnání množin a výsledky celého procesu synchronizace na straně serveru,
- ošetření aplikace – odpověď serveru i v případě interní chyby.

Ve druhé části, kdy byla otestována korektnost synchronizace dat mezi více klientskými aplikacemi (desktopová aplikace, Android aplikace) používané jedním uživatelským identifikátorem *user ID*, bylo testování rozděleno do následujích kroků:

- vytvoření požadavků – sestavení konkrétního požadavku na synchronizaci,
- zpracování odpovědi – případné uložení a smazání dat po synchronizačním procesu,
- komunikace s databází – ověření správné funkcionality především při práci se šifrovanou SQLite databází,
- ošetření aplikace – při vnitřní chybě, ztrátě internetového spojení atd.

Během testování se projevily některé drobné implementační chyby a nedostatky aplikace, které byly následně opraveny a znovu otestovány. Po fázi testování je možné konstatovat, že aplikace plní body zadání a veškeré funkční požadavky včetně synchronizace dat s každým zařízením zvlášť.

6 Možnosti rozšíření

Aplikaci lze použít jako tzv. *Task manager* – neboli správce úkolů z Activiti *workflow*. Tím je splněn požadovaný cíl práce. Během návrhu a vývoje byla ovšem zohledňována možnost následného rozšíření stávající funkcionality a bezpečnosti aplikace.

První možnost rozšíření je zabezpečení komunikace mezi serverem a klientskými aplikacemi. V této práci byl kladen důraz na bezpečné uložení dat v mobilní aplikaci. Komunikace se serverem je realizována zabezpečenou firemní VPN sítí. Při využití aplikace ve veřejné síti by ovšem bylo zabezpečení komunikace, například protokolem SSL [5], nezbytné.

Dalším projektem by mohl být návrh a implementace řešení lokálního zabezpečení dat aplikace. V aktuální implementaci je využita knihovna SQLCipher, jež vyžaduje pro práci s databází přístupové heslo. To je ovšem v projektu součástí zdrojových kódů. Pokud by byl instalační soubor dekompilován, bylo by útočníkem přístupové heslo získáno. Na první pohled snadné řešení je zasílání hesla ze serveru, což ovšem nelze realizovat při off–line režimu. Další možností je například vyžadovat heslo pro přístup do databáze při spuštění aplikace. Tato možnost by ovšem vyžadovala redundantní zadávání hesla jak pro přístup k databázi, tak pro samotné on–line ověření klienta (hesla sice mohou být shodná, ale na rozdíl od hesla pro přístup k databázi se heslo pro ověření může kdykoliv změnit). Z těchto důvodů je aktuální implementace z hlediska bezpečnosti dostačující a další případná modifikace záleží na reakci klientů při testování aplikace a zvážení ostatních rizik.

Také vzhled samotné klientské aplikace by měl navrhnout profesionální grafik mobilních aplikací. Součástí úpravy vzhledu by měla být podpora všech dostupných mobilních zobrazovacích rozlišení. Aktuálně je zobrazení optimalizováno pro rozlišení 480 x 800 obrazových bodů, konkrétně pro mobilní zařízení Samsung Nexus S.

Dále lze k serveru případně připojit jiný datový zdroj bez nutnosti velkých zásahů do projektu. Díky centrálnímu prvku (serveru), přes který všechna data putují, lze například po synchronizaci dat k případným *úkolům* získat doplňující data z Lotus Notes databáze. Ke konkrétnímu *úkolu* doplňující data přidat ve formě formulářové vlastnosti a odeslat klientské aplikaci. Ta zůstane díky mechanismu zobrazení a zpracování formuláře beze změny. Na serveru stačí doprogramovat modul pro komunikaci s jiným úložištěm, následně je tím umožněno případně k datům z Activiti databáze přidat odpovídající data z jiného datového úložiště a klientům odeslat kompletní data v předem definovaném formátu.

7 Závěr

Diplomová práce svým obsahem řeší problematiku správy vybraných požadavků z Activiti *workflow* pomocí webové služby a centrální synchronizace dat. Teoretická část práce (kapitola 2) popisuje základní funkce Activiti *frameworku* a možnosti komunikace s *workflow*. Následující kapitola 3 seznamuje s přístupem k návrhu aplikace. Samotná implementace dvou klientských aplikací a serveru, jenž poskytuje vlastní webovou službu, je popsána kapitolou 4.

Z počátku bylo nezbytné se nejprve seznámit s Activiti *workflow*, jeho základní funkcionalitou, správou, konfigurací a také s aktuálním stavem jeho využití. Po zhodnocení situace a získání základního přehledu o *frameworku* bylo potřebné zjistit možnosti poskytovaného Java a REST API.

Následujícím krokem se stala tvorba úvodního návrhu aplikace, při které byly, po několika konzultacích se zadavatelem práce, určeny základní dostupné implementační technologie a postupy.

Detailní návrh aplikace byl několikrát upravován dle potřeb zadavatele po vzájemné konzultaci. Výsledkem bylo navržení správy požadavků skrze klientské aplikace, které nekomunikují přímo s Activiti *workflow*, ale s vlastním serverem, který zprostředkovává komunikaci s *workflow*, synchronizuje klientská data a nabízí více možností případného rozšíření. Komunikace byla zvolena přes webovou službu, konkrétně navržením a využitím REST rozhraní (2.6.2). Požadavkem na klientskou aplikaci bylo také možnost jejího využití bez připojení k síti (v tzv. *off–line* režimu). Tento požadavek přinesl nutnost navrhnout způsob uložení dat v klientských aplikacích a postup synchronizace dat mezi serverem a jednotlivými klienty. Kompletní postup návrhu, důvody jeho úprav, návrh databáze a výsledný návrh aplikace jsou popsány v kapitole 3.

V pvní fázi implementace byla vyvinuta serverová část (4.1) společně s desktopovým klientem, která primárně slouží k testování komunikace a nových metod zpracování dat (4.2). Postup implementace serveru a datová synchronizace jsou popsány v kapitole 4.1.4. Po úspěšné implementaci serveru a desktopové aplikace byl zahájen vývoj mobilní aplikace, která již mohla být testována oproti vytvořenému serveru.

Mobilní aplikace byla vyvíjena jako jednouživatelská. Zde jsou data ukládána v interní nezabezpečené SQLite databázi. Z tohoto důvodu bylo nutné lokální data šifrovat prostřednictvím SQLCipher knihovny. K této klientské aplikaci byl vytvořen uživatelský manuál, který je přímo součástí mobilní aplikace a tvoří jednu z příloh této práce (7).

Při implementaci aplikace byla snaha o využití jak moderních postupů, které nabízí dnešní frameworky a vývojová prostředí, tak o implementaci návrhových vzorů nebo jejich částí (4.2.6 a 4.2.7).

Konečným krokem se stalo testování aplikace v reálném prostředí (5), při kterém se objevily drobné nedostatky, především ve smyslu ošetření aplikace na výjimky určitých testovacích scénářů. Všechny zjištěné nedostatky byly odstraněny a aplikace byla upravena do finální podoby.

Kapitola 6 popisuje případné návrhy na rozšíření aplikace. Především díky návrhu aplikace je možné rozšíření někola způsoby, jež by se mohly stát některým z následujících zadání projektu.

Součástí přílohy práce je nejen uživatelská příručka, ale také ukázka synchronizačního požadavku, CD s projektem serveru a obou klientkých aplikací včetně instalačního apk souboru. Dále jsou přiloženy SQL skripty pro vytvoření databáze a kompletní dokumentace ke zdrojovým kódům ve formě webových stránek.

Především úspěšnou implementací byla demonstrována možnost využití Activiti *frameworku* s jistotou zachování konzistence dat. Výhodou této aplikace je v neposlední řadě také možnost správy požadavků mobilním zařízením, které se stává zcela běžnou součástí našeho profesního i soukromého života.

Práce byla vyvíjena podle specifikací společnosti Actis s.r.o. Při vývoji aplikace bylo využito několik open–source knihoven, proto je i samotná výsledná aplikace zveřejněna pod open–source licencí MIT [13].

Reference

[1] JACKSON CODEHAOUS. *High-performance JSON processor*. [online] 2013 [cit. 2013-03-15]. Dostupné z: `http://jackson.codehaus.org`.

[2] D. CROCKFORD. *The application/json Media Type for JavaScript Object Notation (JSON)*. [online] 2006 RFC editor [cit. 2013-02-17]. Dostupné z: `http://www.rfc-editor.org/rfc/rfc4627.txt`.

[3] ROY THOMAS FIELDING. *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation [online] 2000 [cit. 2013-04-11]. Dostupné z: `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

[4] THE APACHE SOFTWARE FOUNDATION. *Apache License Version 2.0*. [online] 2012 [cit. 2012-11-14]. Dostupné z: `http://www.apache.org/licenses/LICENSE-2.0.html`.

[5] A. FREIER a et. *The Secure Sockets Layer (SSL) Protocol Version 3.0*. RFC editor. [online] 2013 [cit. 2013-04-03]. Dostupné z: `http://tools.ietf.org/html/rfc6101`.

[6] GOOGLE. *Android API Guides*. [online] 2013 [cit. 2013-03-25]. Dostupné z: `http://developer.android.com/guide/components/index.html`.

[7] GOOGLE. *Android API Guides - Tasks and Back Stack*. [online] 2013 [cit. 2013-04-03]. Dostupné z: `http://developer.android.com/guide/components/tasks-and-back-stack.html`.

[8] GOOGLE. *Google Play*. [online] 2012 [cit. 2013-03-28]. Dostupné z: `http://play.google.com`.

[9] GOOGLE. *Managing the Activity Lifecycle*. [online]. 2013 [cit. 2013-08-05]. Dostupné z: `http://developer.android.com/training/basics/activity-lifecycle/index.html`.

[10] GOOGLE. *Platform Versions dashboards*. [online]. 2013 [cit. 2013-08-05]. Dostupné z: `http://developer.android.com/about/dashboards/index.html`.

[11] OBJECT MANAGEMENT GROUP. *Business Process Model And Notation (BPMN) Version 2.0*. [online] 2013 [cit. 2013-03-25]. Dostupné z: `http://www.omg.org/spec/BPMN/2.0`.

[12] IBM. *IBM Lotus Notes*. [online]. 2013 [cit. 2013-08-05]. Dostupné z: `http://www-03.ibm.com/software/products/us/en/ibmnotes`.

[13] OPEN SOURCE INITIATIVE. *The MIT License (MIT)*. [online]. 2013 [cit. 2013-08-05]. Dostupné z: `http://opensource.org/licenses/MIT`.

[14] FELIPE MARQUEZ. *MySQL to SQLite*. [online] 2013 [cit. 2013-02-17]. Dostupné z: `http://mysql2sqlite.felipemarques.com.br`.

[15] MyDevNotes. *JSON vs XML – Part 1: Data Size*. [online]. 2013 [cit. 2013-08-05]. Dostupné z: `http://xphone.me/devnotes/2011/02/json-vs-xml-part-1-data-size`.

[16] ORACLE. *Glassfish - Open Source Application Server*. [online] 2012 [cit. 2013-03-01]. Dostupné z: `http://glassfish.java.net`.

[17] ORACLE. *Jersey*. [online] 2013 [cit. 2013-03-15]. Dostupné z: `http://jersey.java.net`.

[18] ORACLE. *MySQL*. [online] 2013 [cit. 2013-02-17]. Dostupné z: `http://dev.mysql.com`.

[19] RUDOLF PECINOVSKÝ. *Návrhové vzory. Vyd. 1.* Brno: Computer Press, 2007, s. 527. ISBN: 978-80-251-1582-4.

[20] TIJS RADEMAKERS, TOM BAEYENS a JORAM BARREZ. *Activiti 5.10 User Guide*. [online] 2013 [cit. 2013-03-25]. Dostupné z: `http://activiti.org/userguide/index.html`.

[21] *SQLITE*. [online] 2013 [cit. 2013-02-17]. Dostupné z: `http://www.sqlite.org`.

[22] TUTORIALSPOINT. *Java programming*. [online]. 2013 [cit. 2013-08-05]. Dostupné z: `http://www.tutorialspoint.com/java/java_string_equals.htm`.

[23] LARS VOGELLA. *Tutorials about Android*. [online]. 2013 [cit. 2013-22-04]. Dostupné z: `http://www.vogella.com/android.html`.

[24] W3C. *SOAP*. [online]. 2013 [cit. 2013-08-05]. Dostupné z: `http://www.w3.org/TR/soap`.

[25] ZETETIC. *SQLCipher*. [online] 2013 [cit. 2013-03-25]. Dostupné z: `http://sqlcipher.net`.

[26] ZETETIC. *SQLCipher for Android Application Integration*. [online] 2013 [cit. 2013-03-25]. Dostupné z: `http://sqlcipher.net/sqlcipher-for-android`.

Příloha A - Uživatelský manuál - Activiti mobile v1.0

Activiti Mobile je aplikace pro správu požadavků z Activiti *workflow* prostředí. Umožňuje spravovat úkoly a startovat nové procesy na základě procesní definice uložene v Activiti databázi. Dále nabízí možnost zpracování úkolů v off-line režimu a následnou synchronizaci se serverem. Pro bezpečnost lokálních dat je využito 256-bitové AES šifrování databáze. Aplikace komunikuje se serverem a před prvním použitím je nutné postupovat dle kroků doporučených dodavatelem aplikace (případně konfigurovat vlastní webovou službu).

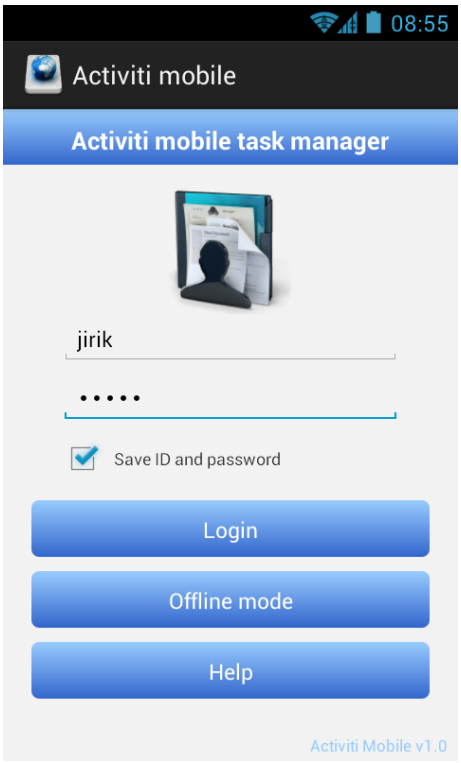
1.1 Instalace aplikace

Aplikace je vytvořena pro mobilní platformu Android 2.2+ a je dostupná pro cílové uživatele ve formě instalačního apk souboru. Apk soubor je nutné nahrát do paměti telefonu a nainstalovat. Po úspěšné instalaci aplikace, která je automaticky umístěna v paměti telefonu, je možné ji přes správce aplikací přesunou na SD kartu.

1.2 Úvodní nastavení aplikace

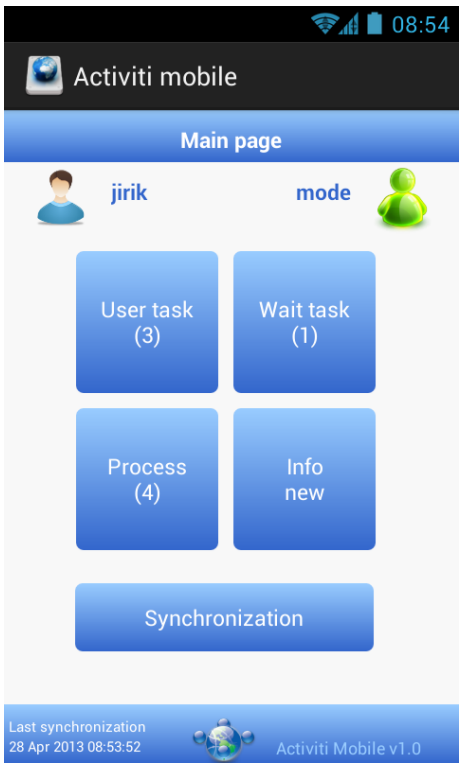
Při prvním spuštění aplikace je nutné ověření uživatele. To je realizováno po vyplnění uživatelského jména (UserID) a hesla (Password). Tyto informace Vám budou zaslány po zažádání dodavatelem aplikace. Úvodní nastavení aplikace obsahuje informace o připojení k hlavnímu serveru. Po zaškrtnutí „Save ID and password“ a úspešném přihlášení se při dalším použití aplikace tato data automaticky předvyplní. Offline mode je umožněn až po úspěšném prvním přihlášení na server. Případná změna konfigurace připojení -> Menu -> Settings -> Mobile server URL.

Po prvním přihlášení je aplikace registrována z důvodu bezpečnosti na konkrétní „UserID“, od této chvíle ji může využívat pouze uživatel se správným „UserID“.



Obrázek 1: Přihlašovací obrazovka

1.3 Použití aplikace



Obrázek 2: Hlavní obrazovka

Po úspěšném prvním přihlášení je zobrazena základní obrazovka aplikace s informací o uživateli a stavu aplikace (mode).

- Zelený panáček – online režim, proběhla úspěšná autentizace a je zpřístupněna synchronizace dat
- Červený panáček – offline režim, je umožnena práce pouze s lokálními daty, bez možnosti synchronizace dat

Hlavní rozcestník aplikace tvoří 4 základní dlaždice a tlačítko pro synchronizaci dat se serverem, které je dostupné dle aktuálního režimu. Dále je v levém dolním rohu zobrazeno datum a čas poslední synchronizace se serverem.

1.3.1 Správa procesů

Na této obrazovce se nachází seznam procesů, které má uživatel právo nastartovat. Při volbě procesu je zobrazen datail procesu a možnost nastartovat nový proces. Pozn. příznak „Start in next synchronization“ značí počet startů konkrétního procesu při příští synchronizaci.

1.3.2 Správa úkolů

Na obrazovce (User task) jsou v seznamu zobrazeny vsechny úkoly, které uživatel doposud neřešil. Kliknutím na úkol lze zahájit řešení úkolu.

Při volbě na hlavní obrazovce čekající úkoly (Wait task) jsou zobrazeny úkoly, které uživatel řešil a čekají na dokončení. Tyto úkoly se před dokončením stále dají modifikovat.

Po zahájení řešení úkolu je zobrazen detail konkrétního úkolu a možnost jeho dokončení.

Při dokončení úkolu (Complete task) je dynamicky vygenerován případný formulář s doplňujícími informacemi. Pole označená hvězdičkou jsou povinná.

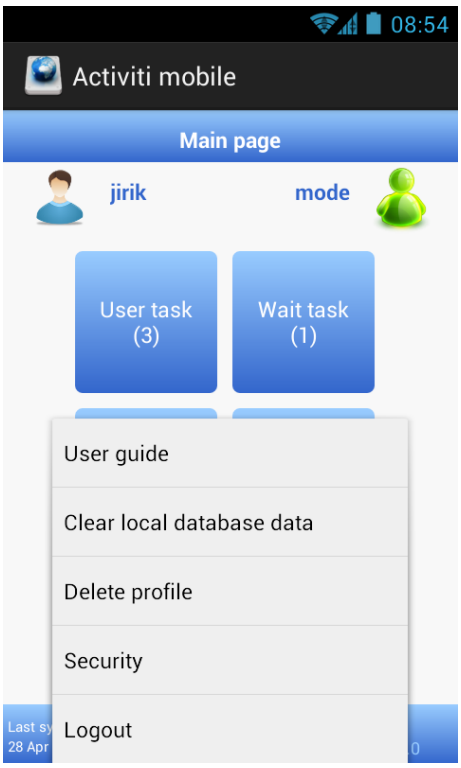
1.3.3 Možnosti aplikace

Na obrazovce (User task) jsou v seznamu zobrazeny vsechny úkoly, které uživatel doposud neřešil. Kliknutím na úkol je možné zahájit řešení úkolu.

Při volbě Wait task (čekající úkoly) jsou zobrazeny úkoly, které uživatel řešil a čekají na dokončení. Tyto úkoly je možné před dokončením stále modifikovat.

Po zahájení řešení úkolu je zobrazen detail konkrétního úkolu a možnost jeho dokončení.

Při dokončení úkolu (Complete task) je dynamicky vygenerován případný formulář s doplňujícími informacemi. Pole označená hvězdičkou jsou povinná.



Obrázek 3: Hlavní nabídka

Aplikace je vytvořena jako jednouživatelská (z důvodu bezpečnosti), pokud je registrována na jednoho uživatele, není možné ji použít uživatelem jiným. Je ovšem možné aplikaci konfigurovat pro případné smazání uživatelské registrace. Aplikace umoňuje následující nastavení v menu základní obrazovky.

- User guide – zobrazí tento manuál.
- Clear local database data – v případě chyby nebo pro případné zvýšení bezpečnosti vymaže všechna data v lokalní databázi, po synchronizaci jsou data opět uložena. Uživatel zůstane registrován v aplikaci.
- Delete profile – smaže kompletní nastavení a data aplikace. Je možné využít aplikaci jiným uživatelem.
- Security – umožňuje zapnout nebo vypnout ověření uživatele při použití off–line režimu.
- Logout – odhlásí uživatele z aplikace.

1.4 Odinstalace aplikace

Aplikaci lze odinstalovat klasickým zbůsobem ve správci aplikací v systému Android.

1.5 Technické specifikace aplikace

- Interní uložení dat – SQLite, 256-bit AES šifrování
- Minimální verze systému – Android 2.2 Froyo
- Minimum API Level – 8
- Target API level – 17
- Rozlišení optimalizováno pro – Nexus S 480x800

Příloha B - ukázka synchronizačního požadavku

Synchronizační požadavek (task)

Požadavek na synchronizaci *úkolů* má následující atributy:

- LoginInfo userInfo – informace u uživateli, který o synchronizaci žádá (ID a heslo),
- ArrayList<RESTTAsk> userTask – seznam identifikátorů *úkolů*, které má uživatel ve své databázi.

Následně je provedena synchronizace na straně serveru, po této akci je vrácena odpověď.

Synchronizační odpověď (task)

Synchronizační odpověď má násedující atributy:

- int status – informace výsledku synchronizace, případné chyby,
- UserTask[] new_tasks – kompletní nové objekty, které si klient uloží ve své databázi,
- RESTTask[] old_tasks – identifikátory *starých* dat, která klientská část smaže ze své databáze,
- TaskForm[] new_forms – nové formuláře k novým *úkolům*, které budou uloženy do klientské databáze.

Informace o *starých* formulářích není potřeba zasílat. Každý formulář je v databázi navázán na ID *úkolu*. Pokud je *úkol* smazán, jsou kaskádově smazány i veškerá formulářová data (pole, enumerátory).

Příloha C - přiložené CD

Na přiloženém CD se nachází:

- DP_Jiri_Sojka.pdf – tato práce ve formátu pdf,
- server.zip – projekt serverové části aplikace,
- client_desktop.zip – projekt desktopové klienstké části aplikace,
- client_android.zip – projekt mobilní klientské části aplikace,
- database_script.zip – soubor se skripty pro vytvoření serverové a klientské databáze,
- doc.zip – soubor s dokumentací ke zdrojovým kódům,
- Mobile_client.apk – instalační soubor klientské aplikace pro platformu Android.